HELSINKI SCHOOL OF ECONOMICS (HSE)
Department of Management

HIGH-LEVEL PROCESS MODELS AND GAME DEVELOPMENT

-Case Zobmondo!!

Information Systems Science
Master's thesis
Sanna Laukkanen 70221-6
Spring 2003

Approved by the Head of the Department of Management 4 / 4 2003, and

awarded the grade 90 p.

MATTI ROSSI KTI

VIRPI TUUNAINEN KTI

ABSTRACT

Computer applications and hence, software, are becoming ever more critical to modern society. The significance of a high-quality software and thus, high-quality software process has given raise to process modeling, which can be seen to play a central role in software process research. The importance of software process modeling is being increasingly acknowledged also in game development, which is an area of software development strongly growing in both size and importance. Interestingly, however, research on game development is, for the time being, nonexistent.

One of the objectives of this study is the development and piloting of the framework that can be used for software process analysis in terms of high-level process models (also known as life cycle models) and support areas. This framework can be used by researchers as well as practitioners, and in game development as well as in any other software development. By developing this framework based on a comprehensive review on the common high-level process models this study also aims to begin consolidating the theoretical basis for the high-level process models, which until now has remained incoherent.

The study also aims to help establish basis for the further game development research by bringing out the growing importance and complexity of the game development processes and by conducting a pilot case study in this area. In the case study two game development processes related to the mobile game Zobmondo!! at Codetoys Inc. are retrospectively analyzed with the framework developed in this study. The specific research questions addressed by the case study are "what are game development processes like in terms of high-level process models?" and "what are the needs of game development processes in terms of support areas?".

The results of the case study are consistent with the previous discussion on the high-level process models, that is, there is a homeground for both experimentation-oriented and orderly planning-oriented software process approaches. Moreover, in situations where creativity is required, rigidity in software process definition should be avoided in order to make flexible tailoring of the software process to the unique needs of the project possible.

Keywords:
Software development, game development, process models, life cycle models

TIIVISTELMÄ

Tietokoneohjelmistot näyttelevät yhä kriittisempää roolia jokapäiväisessä elämässämme. Korkealaatuisten ohjelmistojen ja täten korkealaatuisten ohjelmistotuotantoprosessien tarve on nostanut keskeiseen asemaan ohjelmistoprosessien mallintamisen. Ohjelmistoprosessien mallintamisen tärkeys on kasvavassa määrin huomattu myös pelituotannossa, joka on ohjelmistotuotannon kasvava osa-alue.

Tämän tutkielman tavoitteena on luoda viitekehys, joka mahdollistaa ohjelmistoprosessien analysoimisen vaihejakomallien ja ohjelmistoprosessin tukea vaativien osa-alueiden avulla. Viitekehys on tarkoitettu työkaluksi niin tutkijoille kuin ohjelmistotuotantoalalla työskenteleville, sekä soveltuvaksi niin pelien kuin muidenkin ohjelmistojen tuotantoprosessien analysoimiseen. Viitekehys perustuu kattavaan vaihejakomalleja koskevaan kirjallisuuskatsaukseen, ja tämän pohjalta tehtyyn tutkielmassa esiteltävään vaihejakomallien esittelyyn, jonka kautta tutkielma pyrkii vahvistamaan ohjelmistotuotannon vaihejakomallien tähän asti epäyhtenäisenä pysynyttä teoriapohjaa.

Edellä mainittujen tavoitteiden lisäksi tutkielma pyrkii luomaan perustaa tulevalle pelikehitystutkimukselle tuomalla esiin peliteollisuuden jatkuvasti kasvavan merkityksen sekä pelien kehittämiseen liittyvien ohjelmistoprosessien kasvavan kompleksisuuden. Tutkielma esittelee tapaustutkimuksen, jossa tutkielmassa kuvattua viitekehystä käyttäen tarkastellaan retrospektiivisesti kahta Zobmondo!! mobiilipelin kehittämiseen liittyvää prosessia Codetoys Oy:ssa. Keräämällä empiiristä aineistoa pelikehitysalalta, jonka tutkimus on vielä toistaiseksi ollut vähäistä, tutkielma pyrkii toimimaan eräänlaisena pilottina ja luomaan perustaa tulevalle tutkimukselle tällä kiinnostavalla ja merkittävällä ohjelmistotuotannon osa-alueella.

Tapaustutkimuksen tulokset ovat yhteneväiset vaihejakomalleihin liittyvän aiemman tutkimuksen kanssa. Niin suunnitelmalliset kuin lyhytjänteisemmät kokeiluun perustuvat ohjelmistokehityksen lähestymistavat ovat tarpeen, jotta eri tekijöiden ohjelmistoprosesseille asettamiin vaatimuksiin voidaan tarpeen mukaan tehokkaasti vastata. Erityisesti sellaisissa ohjelmistoprosesseissa, joissa prosessin eri toimijoiden luovuus näyttelee keskeistä roolia, tulee prosessimäärittelyn tiukkuutta välttää, jotta joustava ja tehokas reagoiminen kulloinkin esiin nouseviin tarpeisiin olisi mahdollista.

Avainsanat:
Ohjelmistokehitys, ohjelmistotuotanto, ohjelmistoprosessit, pelikehitys, prosessimallit, vaihejakomallit

CONTENTS

1 INTRODUCTION

1.1 Background of the study

"Software process research deals with the methods and technologies used to assess, support, and improve software development activities (Fuggetta 2000)."

Both individual and organizational activities involve ever more computers (Lehman 1998). As it is software that delivers the computing potential embodied by computer hardware, the growing dependency on software becomes evident. In addition to being ever more critical to modern society, the applications of computers and hence, software, are becoming increasingly diverse and complex (Fairley 1985, p. 2). It has been even argued that measured in structure, content, and functionality, software systems are by far the most complex artifacts mankind has ever created (Lehman 1998). Combined with the premise that high-quality production process is on of the prerequisites for a high-quality product (Chroust 1996; Pressman 2000, p. 45; Sommerville 2001, pp. 536, 560) these viewpoints strongly emphasize the importance of the software development process (hereinafter software process), and especially, its management and improvement (Lehman 1998).

Both software process improvement and management have been recognized to benefit from software process definition (Curtis, Kellner et al. 1992). The software process definition can focus on studying the existing software processes (descriptive), that is, how software actually is developed, or defining the desired software processes (prescriptive), that is, how software should be developed (Lonchamp 1993). The actual or desired software process is expressed and studied through more or less formal techniques for increased understanding, communication, education, process execution support, process management, comparisons, impact analysis, reuse, and standardization (Lonchamp 1993), thereby helping the organization in search of the quality software process.

The expressing and studying of the defined software process can be aided by software process modeling. As models, in general, are abstractions of reality highlighting those features of the world which are relevant to the goals of the modeler (Derniame, Badara et al. 1999, p. 165), a software development process model (hereinafter process model) can be considered as an abstract software process description expressing a particular view on the process (Lonchamp 1993). A given process model can be represented at different levels of abstraction (Lonchamp 1993) in order to describe the phenomena being modeled in more or less detail. Models of

low abstraction level can be used to convey details such as inputs and outputs of activities, flows of artifacts, constraints, or resources used (Lonchamp 1993). High-level models, then, can be modified into more detailed models by gradually refining and adapting them to a specific domain or usage (Lonchamp 1993). The notation used in the representation of a process model can be more or less formal, ranging from a precisely defined formal syntax and semantics to natural language (Derniame, Badara et al. 1999, p. 27).

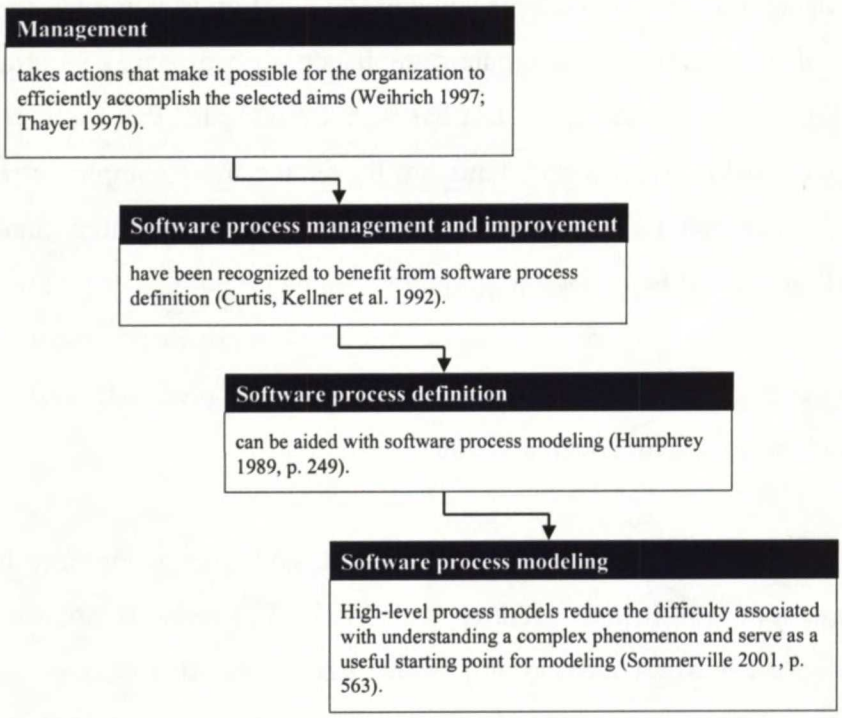


Figure 1. Relationships between software process management, improvement, definition and modeling based on (Humphrey 1989, p. 249; Curtis, Kellner et al. 1992; Weihrich 1997; Thayer 1997b; Sommerville 2001, p. 563).

In fact, the optimal level of formalism in representing the process models has been a subject of vivid discussion in software process research. Informal representation techniques are considered to suffice for communicational purposes and conceptual models, but more rigorous, formal representation techniques have been proposed as a prerequisite for using the models as a basis for precise and reliable analysis or process automation (Curtis, Kellner et al. 1992; Osterweil 1997), and to assure the quality of products to be developed (Osterweil 1997). A formally defined syntax and semantics would also make the process model executable by computer and thus, help, for example, process simulation (Kellner 1988). Moreover, a computer interpreted process model guiding the participants through the software process could help the enforcement of the conformity of an actual software process with the

prescriptions (Curtis, Kellner et al. 1992; Chroust 1996). However, an important limitation to the formal approach is the fact that modeling a problem precisely requires that the problem domain is fully understood (Lehman 1987). This, of course, is often not the case with software processes and consequently, the answer to the question of optimal level of formality can not be given at the universal level but depends on the situation.

Consequently, it can be stated that the significance of a high-quality software process has given raise to process modeling, which, indeed, can be seen (Iivari 1991; Curtis, Kellner et al. 1992; Yu and Mylopoulos 1994) to play a central role in software process research. Further, it has been recognized by both researchers and practitioners that software process definition and modeling can not be a one-shot undertaking after which the process is frozen for good (Fuggetta 2000). The organization's way to develop software must be continuously assessed and improved to increase the organization's ability to successfully deal with the changing circumstances (Fuggetta 2000). These observations have motivated a range of initiatives devoted to the creation of methods for systematic software process improvement and assessment (Fuggetta 2000). Examples of these are Capability Maturity Model (CMM) and its European counterparts Bootstrap and SPICE developed to help the organizations assess the maturity level of their software processes and determine the priority of various actions needed in improvement (Humphrey 1989; Derniame, Badara et al. 1999, p. 1).

As for this study, the purpose is to contribute to software process research by providing a comprehensive synthesis of the high-level process models (also known as life cycle models) presented in the software development literature. High-level process models are very abstract, generic process models which primarily describe the order of fundamental process steps involved in the software development (Lonchamp 1993), thereby helping the participants of the software process become aware and gain an increased understanding of the process in question (Derniame, Badara et al. 1999, p. 4). As for formality, these models are considered semi-formal, in other words, they are represented in some kind of graphical notation (Lonchamp 1993). Through their low level of detail high-level process models reduce the difficulty associated with understanding a complex phenomenon and thus, serve as a useful starting point for the software process definition (Sommerville 2001, p. 563). As for the scientific importance of the high-level process models, this has been acknowledged by, for example, Iivari (1991) who notes that these models are one of the major contributions of the software engineering school, one of the contemporary schools of information systems

development. Interestingly, the previous research on software processes seemed to be lacking a comprehensive review of the different high-level process models. This study attempts to begin filling this gap in the research.

Further, this study attempts to examine software processes in a Finnish mobile game development company. Game development is chosen as a topic for two reasons. First, as a result of the growing importance of games industry, game development projects are evolving from small-team efforts to industrial size projects and the complexity of game development processes is growing dramatically (Walton 1998; vom Scheidt 2000). Partly resulting from this, the games industry has been experiencing the typical process related problems of software development (Walton 1998; vom Scheidt 2000). As the previously unorganized software processes that took place in garages need increasingly be replaced by professional, large-scale game development, there is a clear demand for software process definition and modeling also in the games industry. Second, although there is a lot of previous research on the games' social, psychological, cultural, and health impacts, research on the game development seems, interestingly, to be only taking shape. To this early stage of the game development research this study aims to contribute by giving an overview of the game development processes in a Finnish mobile game development company.

1.2 Research objectives

The objectives of this study are as follows. The objective of the theoretical part of this study is threefold. First objective is to provide a comprehensive review of the high-level process models presented in the software development literature by discussing their characteristic features and implications for the underlying software processes. Furthermore, the nature of the support provided by these models to software process is analyzed. The second objective of the theoretical part, then, is to construct a framework for positioning different high-level process models and for analyzing software processes in terms of high-level process models and support areas. Finally, the third objective is to give an introduction to game development in general and thereby to help comprehend the context of the empirical part of this study.

The objective of the empirical part, in turn, is to explore the game development processes in a Finnish mobile game development company with the help of framework developed in the

theoretical part of the study. To be more specific, the empirical part of this study aims to answer the following questions:

- What are game development processes like in terms of high-level process models?
- What are the needs of game development processes in terms of support areas?

For the time being the research on the game development as software process is, practically, nonexistent. Further, although much has been written about the high-level process models, the theoretical ground related to them has remained incoherent. Consequently, the nature of this study is explorative. The overall aims are to help establish a basis for further research in game development through conduction of pilot case study in the area of game development processes, but also to begin consolidating the theoretical basis for the high-level process models. The latter objective is accomplished by the comprehensive review of the different high-level process models and the development of a preliminary framework for positioning these models.

1.3 Research methodology

The section of the theoretical part discussing high-level process models is based on a literature review and presents a synthesis of the contributions of many different authors. The review focuses on often-referred academic papers published in recognized academic journals and conference proceedings, but covers also the well-known textbooks on the topic. For the purposes of this study, the textbooks provided a condensed and systematized overview of the software engineering school today, but more importantly, led to the origins of the software process research. As for the scope of the literature review, this was outlined by examination of new references until no new relevant data emerged regarding the high-level process models. As to the section of the theoretical part introducing game development, this is based on game development practitioner literature and online resources, as academic research on game development is, for the time being, nonexistent.

The empirical part, then, is based on a case study in a Finnish mobile game development company. Case study was chosen as a research method as it is the preferred strategy when (Benbasat, Goldstein et al. 1987; Yin 1990, pp. 13-17):

- subject researched does not enjoy a strong theoretical base and exploratory research questions like the ones addressed by this study (see section 1.2) are being posed,

- control of the actual behavioral events in the processes researched is not necessary for the purposes of the study,
- focus of the study is on a complex, non-historical phenomenon.

The conduction of the case study and the framework used in the analysis of the results are discussed in more detail in chapters six and seven.

1.4 Outline of the study

In the following chapter the concept of software and fundamental software development activities are first introduced. After this the importance of a defined software process is discussed. In chapter three the common high-level process models and their characteristic features are examined and the implications of these models for the underlying software processes are discussed. Chapter four, then, outlines the application of and presents a summary of the discussed high-level process models.

Chapter five serves as an introduction to the game development and thus, helps comprehend the context of the empirical part of this study. The research framework and methodology applied in the empirical part of the study are introduced in chapters six and seven, and the case study conducted is discussed in chapter eight. Finally, in chapter nine the findings of the study are discussed and the conclusions are drawn.

2 SOFTWARE PROCESS

2.1 Defining software

According to IEEE (IEEE Std 610.12 1990), software refers to computer programs, procedures, and associated documentation, and data pertaining to the operation of a computer system. Computer programs are a combination of computer instructions and data definitions enabling computer hardware to perform computational or control functions. Procedure, in turn, can be defined as a course of action to be taken in order to perform a given task, but also as a portion of a computer program that performs a specific action. Finally, documentation is a collection of documents on a given subject, for example, on how a product is to be used, or maintained. (IEEE Std 610.12 1990)

A software system, on the other hand, is a collection of interacting programs, coordinated in function and disciplined in format, constituting a composition for accomplishing larger tasks (Brooks 1982, p. 6).

2.2 Categorizing software

A term software product can be used to refer to the complete set of computer programs, procedures, and possibly associated documentation and data designated for delivery to a user (IEEE Std 610.12 1990). Sommerville (2001 p. 5) divides software products in two categories. The first category is called generic products, sometimes referred to as shrink-wrapped software. These are produced by a development organization and sold on the open market to any customer who is able and willing to buy them. Examples of this type of a software product include word processors and spreadsheets. The second category is called customized (also known as "bespoke") products. These, in turn, are software products, which are commissioned by a particular customer. Consequently, in contrast to generic products, this type of software product is developed specially for a particular customer. One example of this is software developed to support a particular business process.

As to the differences between the two abovementioned categories, an important question from the developer's point of view is who controls the software process. In generic products the organization responsible for the software development controls the software requirements specification and then sells the resulting product to the interested customers on the open

market. For custom products, on the other hand, the customer buying the software is known beforehand and thus, the requirements specification is usually developed and controlled by the customer. (Sommerville 2001, pp. 5-6)

Software products can also be categorized according to application areas. The following typology of software applications is an adaptation of the generic categorizations introduced by Pressman (2000, pp. 9-10) and Haikala (Haikala and Märijärvi 2000, pp. 5-6):

- System software: Collection of programs serving other programs, characterized by heavy interaction with computer hardware, heavy usage by multiple users, resource sharing and multiple external interfaces. This category includes, for example, operating systems.
- Real-time software, reactive systems: This category includes software that monitors, analyzes, and controls real world events as they occur. This type of software is characterized by a real-time response, typically ranging from 1 millisecond to 1 second. The term real-time software is also used to refer to process control and automation systems and embedded systems.
- Business software: Being the largest single software application area this category includes software that, for example, processes business information and automates office routines thereby facilitating business operations and decision-making.
- Engineering and scientific software: These typically assist modeling of natural phenomena and designing of systems, and can be characterized by number crunching algorithms.
- Embedded software, control system: This type of software can be found in intelligent products and is used to control these products. Examples of these are the keypad control for microwave ovens, and fuel control and dashboard displays for modern automobiles.
- Personal computer software: Examples of this type of software are spreadsheets, word processing, entertainment, multimedia, and external network access.
- Web-based software: Web pages retrieved by browser can be considered as software that incorporate executable instructions, for example CGI, HTML, Perl or Java, and data such as hypertext, visual or audio formats.
- Artificial intelligence software, knowledge-based system: This category includes software that make use of nonnumeric algorithms to solve complex problems that are not amenable to straightforward analysis.

For example Pressman (2000, p. 9) has pointed out, that developing meaningful generic categories for software applications is difficult. The dividing lines between different categories disappear as the complexity of software grows (Pressman 2000, p. 9) and consequently, complex software systems often are obscure hybrids having features of many

different software application types (Haikala and Märijärvi 2000, p. 6). Thus, it has been proposed (Haikala and Märijärvi 2000, pp. 6-7) that instead of using these generic categorizations, the following software features should be examined in order to get a better view of the nature of a particular software product:

- Size of the software and the amount of data to be processed: These can be expressed in, for example, bytes, lines of code, amount of functionality delivered, number of screens, size or complexity of the database.
- Response time and real time requirements: The responsiveness of the software to events from its environment. Hard real time requirements indicate that response must be quicker or exactly match the response time requirement.
- Reliability: Quantifies the ability of the software to function with respect to requirements under stated conditions for a specified period of time. If high level of reliability is required also high level of protection against the software failures is required.
- Distribution: The level of distribution of the information processing over several computers.
- Level of customization: Customized versus generic products.

2.3 Generic software process

In this section the fundamental software development steps are introduced. Although the customer is mentioned in many connections throughout the rest of the study, and thus the view of a custom product development is taken, this does not imply that a pre-known customer is necessarily needed. The generic software process for the generic product is the same than for a custom product, only the decisions otherwise made by customer are now made by the developer.

Software is produced through software process by which user needs are translated into a software product. This process involves translating user needs into software requirements, transforming the software requirements into design, implementing the design in code and testing the generated code (IEEE Std 610.12 1990). Also software maintenance (discussed below) is here regarded as a part of the software process. As for the product resulting from the software process, this may be developed from scratch or, what is becoming more and more common, using an existing software system as a basis that is extended and modified into a new software product (Sommerville 2001, p. 43).

As was already discussed above, software processes are complex and dependent on human judgment and creativity. Consequently, single archetypal idea of software process neither suffices nor exists. Different types of software processes do exist in different organizations and for the different types of software. However, there are fundamental software development steps that are typical of the various instances of software process. These are discussed below. (Sommerville 2001, p. 43)

Software process A set of partially ordered *process steps*, with sets of related *artifacts*, human and computerized *resources*, organizational structures and *constraints*, intended to produce and maintain the requested software *deliverables*.	**Process step** a sub-process of the complete process or of any recursively defined sub-process. Aims at reaching a sub-goal of the overall goal.
	Artifact A product created or modified during a process either as a required result of a given process (deliverable) or to facilitate the process.
	Resource An asset needed by a process step to be performed. Includes *agents*, that is, performers of a process that may be humans or computerized tools.
	Constraints Any restriction affecting the performance of a process

Table 1. Definition of the software process according to Lonchamp (1993) and used in this study.

Requirements engineering: In this step the functionality of the software to be produced and constraints on its operation, in other words requirements, are defined (Fairley 1985, p. 40). As software always is only a part of some larger system, requirements for the system software is going to be part of must be taken into consideration (Pressman 2000, p. 27). The requirements engineering results in a document called requirements specification, which then acts as a description for the software to be developed (Sommerville 2001, p. 55). Consequently, the errors made at this phase lead to problems later in the software process (Sommerville 2001, p. 55) and special attention should be paid to defining requirements in a way which reduces the likelihood of misinterpretations and inconsistencies (Pressman 2000, pp. 277-278) and makes it possible to verify whether the software developed conforms to the requirements specification or not (Boehm 1979). An important issue to be considered here is choosing the optimal degree of formality for requirements definition (Boehm 1979).

To be more specific, requirements engineering consists of four main stages. In feasibility study the possibilities of meeting the customer needs using the available technologies are estimated. Also the relevance and cost-effectiveness of the proposed product and budgetary and schedule constraints are considered. In requirements elicitation and analysis the requirements for the product are derived. In addition to discussions and observations this may

involve the development of, for example, models in order to help to elicit the requirements and to eliminate harmful ambiguity. In requirements specification the information gathered in the previous stage is translated into a document defining a set of requirements. Two types of requirements included into this document are user requirements, which are statements of high abstraction level, and system requirements, which are more detailed descriptions of the needed functionality. Finally, in requirements validation the proposed requirements are checked for consistency and completeness. (Sommerville 2001, p. 56)

Software design and implementation: As the question requirements engineering aims to solve is "what the software will do", the question for software design is "how the software will do it". During software design the structure of the software to be implemented, the data, which is a part of the system, and the interfaces between system components, are described based on the requirements specification. This may include revision of the requirements specification. The resulting design description is produced in a sequence of steps, where the output of the previous design stage acts as a specification for the next stage. The general activities in the design process are architectural design, abstract specification, interface design, component design, data structure design and algorithm design. These stages are interleaved, and involve feedback from one stage to another. Hence, design rework is inevitable. (Fairley 1985, p. 40; Pressman 2000, pp. 330-332; Sommerville 2001, pp. 56-58)

The final software design description is a result of an iterative process, where a number of different versions are developed and earlier designs are corrected. A design of higher level of abstraction is gradually transformed to a detailed design of lower level of abstraction. The final results are precise specifications of the algorithms and data structure to be implemented. (Pressman 2000, p. 332; Sommerville 2001, pp. 56-57)

In implementation, software design specifications are converted into an executable code, in other words, into an actual software system. In addition to programming the implementation includes correction of the erroneous code and documentation of the generated code. It is quite normal that the later stages of software design process and implementation are interleaved. (Fairley 1985, p. 40; Sommerville 2001, pp. 59-60)

Software verification and validation: The software must be verified to ensure that it conforms to its specification and validated to ensure that it is suitable to its intended purpose.

Verification and validation involve, for example, reviews (inspections and walkthroughs) at the different steps of software process, and testing, which is carried out after, but also partially in conjunction with implementation. Reviews are a static way of checking and analyzing the software and work with representations such as requirements and design documentation, or source code. Unlike reviews, which don't require the software to be executed, testing is based on the execution of the program code and is needed to demonstrate the operational usefulness of the software. (Sommerville 2001, pp. 60, 420; Schach 2002, pp. 139, 145)

As software systems are, generally, composed of many layers of sub-elements within main-elements, the testing should be carried out in stages proceeding from the component level toward the system level, and ideally, incrementally in conjunction with implementation. During the early testing stages, the developers who generate the code often do the testing. The later stages involving high level of integration among various components, in turn, may involve testing specialists. (Pressman 2000, pp. 426-429, 468; Sommerville 2001, pp. 60-62)

The typical stages in testing process are unit testing, where system components are tested individually; module testing, where collections of dependent components are tested; sub-system testing, where collections of modules are tested; and system testing, where possible errors in interactions between sub-systems are traced. System testing is also about validating that the system as a whole meets its requirements. The stage where the system is no more tested with the simulated test data but with actual data is called acceptance testing (alpha testing). Acceptance testing reveals, for example, problems in requirements definition and unacceptable performance, and it also shows if the system does not meet the customer's requirements. Finally, in beta testing, the software product is exposed to its real use by delivering it to users outside the development organization. (Sommerville 2001, pp. 61-63)

Testing is an iterative activity. As defects are discovered during the testing the program must be debugged, that is, the faulty part of the code must be located and revised. As a result, previous stages in the testing process may have to be repeated in order to ensure that changes made to a program have not introduced new faults into it. (Sommerville 2001, pp. 61, 423)

Software evolution (maintenance): After the software has gone into use, the need for software evolution in order to meet the changing customer needs arises (Sommerville 2001, p. 63). The changes made after delivery are often called software maintenance (Brooks 1982, p. 120; Sommerville 2001, p. 63) and include enhancing the capabilities of a software product, adapting the software product to new processing environments and requirements, and removing defects (Fairley 1985, p. 9). Thus, maintenance can be considered as the alteration and correction of software requirements, design, and code after the software product has been released. Moreover, maintenance can be regarded either as a down-stream development activity of the initial software development process, or as a completely new development process (Fairley 1985, p. 54). Maintenance plays a significant role in software development, constituting 60-90% of the total software development effort for a particular software product (Fairley 1985, p. 9). As maintenance requires thorough understanding of the software product, importance of good and timely documentation (see, for example, Parnas and Clements 1986) and well-structured code as elements of a software product are emphasized (Boehm 1976).

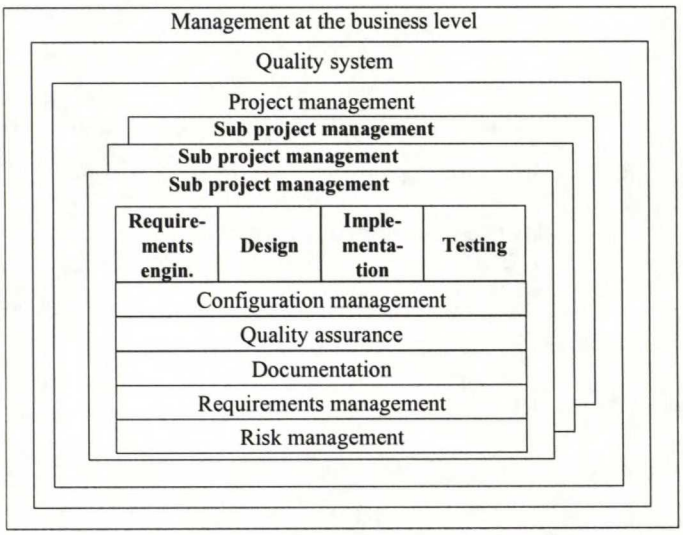


Figure 2. Totality of software development activities (Haikala and Märijärvi 2000, p. 23).

Like any other activities, also the fundamental software development steps do not exist in vacuum, but in parallel with various supporting and enabling factors. The above figure (Haikala and Märijärvi 2000, p. 23) intends to give a rough idea of the totality of activities related to software development.

Apart from supporting and enabling activities, support for software process manifests itself in the form of computer programs called software tools (IEEE Std 610.12 1990). These tools are

often referred to as CASE (computer-aided software engineering) tools (IEEE Std 610.12 1990) and are used to provide automated or semi-automated support in the software process (Pressman 2000, p. 19) and thereby improve quality and productivity of software development work. These tools can be either standalone solutions addressing some particular software development activity without communicating directly with other tools, or integrated to each other so that one tool can use information created by another (Pressman 2000, p. 870) and a comprehensive support environment is accomplished.

As for the different domains of knowledge required in software development, accomplishing software development activities requires software development methods and techniques, that is, guidelines on how to use technology and accomplish development activities effectively. Further, as software development, in general, is carried out by teams of people that have to be coordinated and managed, knowledge on organizational behavior plays an essential role. Finally, in order to be able to address customer's needs and operate in a specific market setting, up-to-date knowledge of marketing, economy and application environment domain is needed. (Fuggetta 2000)

As providing a more detailed, holistic view of software development is not within the scope of this study, the reader is advised to see, for example, (Thayer 1997), where the various aspects of software development are discussed in more detail.

2.4 Defined software process

Software process as a complex entity where intangibility and expected high quality of delivered product together with schedule and budget constraints need special attention, requires effective management (Fairley 1985, pp. 3-5; Bandinelli, Fuggetta et al. 1995; Sommerville 2001, p. 72). Unfortunately, however, the ever-evolving technologies and management practices have not, in many cases, been able to substantially improve the way software development projects are managed and improved (Bandinelli, Fuggetta et al. 1995) and many software development projects have failed in one aspect or another (Neumann 1993; Gibbs 1994; Schmidt, Lyytinen et al. 2001). Combined with the fact that investments in software are continuously increasing worldwide (Bandinelli, Fuggetta et al. 1995) and the premise that the quality of software product is largely determined by the quality of the process used to develop it (Chroust 1996), this has resulted in software process improvement

becoming one of the most important targets for many industrial and research initiatives (Fairley 1985, pp. 4-5; Bandinelli, Fuggetta et al. 1995; Fuggetta 2000).

Central in software process improvement is the idea of software process definition, an explicit description of how software is or should be done (Bandinelli, Fuggetta et al. 1995; Humphrey 1998). All software development organizations despite their different levels of sophistication in mastering their processes follow a process of some kind, implicit or explicit (Derniame, Badara et al. 1999, p. 2). However, it is the software process definition which helps the organizations to develop a comprehensive insight as to how, through their processes, they achieve quality products (Lehman 1997). Thus, in a way, a defined software process provides an organization with a consistent framework for performing the work and improving the way it is done (Humphrey and Kellner 1989). The typical consequence of an undefined software process, in turn, is a significantly increased risk to the software development project in predicting and controlling the critical factors of schedule, cost, scope, and quality (Whitten 1995, p. 11). Clearly, these negative effects emphasize the value of a defined software process for both software process improvement and management.

The journey to a defined software process begins with the selection of an appropriate process model (Humphrey 1989, pp. 247-249; Whitten 1995, p. 19). For the purposes of this study, the process model is defined as an abstract representation of a software development cycle, which starts from the idea conception and ends to the initial release of software (IEEE Std 610.12 1990). Thus, from the process model's point of view, software evolution (which is part of the software life cycle (IEEE Std 610.12 1990)) is considered as a whole new process. To be more specific, the process model encompasses the steps required in the development of a new software product or development associated with the maintenance updates of an existing software product (Fairley 1985, p. 37) and defines the principles and guidelines (for example, temporal and causal relationships) according to which these steps have to be carried out (Fuggetta 2000). Hence, process model supports categorization, control, and execution of the various development steps and, indeed, should be defined for each software development project (Fairley 1985, pp. 31, 37).

The model to be applied in a particular software development project can be selected from the existing process models or a modification or hybrid of the existing models may be applied. Alternatively, a completely new process model can be constructed from scratch to meet the

unique needs of a particular project. All in all, the bottom line is that a process model understood and accepted by all concerned parties improves project communication and comprehension, and enhances project manageability, resource allocation, cost control, and product quality (Fairley 1985, p. 37). In this way it aids decision-making and co-operation between project participants and guides the participants through a complex software process in an orderly way.

The process model may be expressed at a greater or lesser level of abstraction (Lonchamp 1993). The desirable level of detail varies with the purpose for which the model is to be used and the level of expertise of the person using the model as a basis for process execution (Curtis, Kellner et al. 1992). High-level process models (life cycle models) provide an overview understanding of the processes concerned and focus on step sequencing at a high level (Humphrey 1989, p. 249). In these generic models the granularity of the process steps included is quite large. At this very high level of simplification, the processes are very much the same in many different organizations (Sommerville 2001, p. 563), and consequently, same high-level process models can be used in many similar projects and organizations sharing common characteristics (Lonchamp 1993). However, when the fundamental high-level steps are broken into more detail, it becomes clear, that one high-level process model has different instantiations depending, for example, on the type of software being developed and the organizational environment (Humphrey 1989, p. 258; Sommerville 2001, p. 563).

When a software process is, for example, thoroughly improved or an automatization of a specific software process activity is attempted and thus, a deeper and more complete understanding about and analysis of a software process is needed, process models of lower abstraction level become necessary (Humphrey 1989, p. 252). Detailed process models are needed also when the level of process knowledge of the person using the process model as a basis for process execution is low (Curtis, Kellner et al. 1992). Ideally, detailed process models support multiple, complementary viewpoints of the process, that is, in addition to representing what is performed, how, when, by whom and where, they describe the data relevant to the process and objects being produced (Kellner 1988). Apart from these powerful representational capabilities, detailed process models should encompass comprehensive analysis capabilities and a capability to make predictions regarding the effects of changes to a process (Kellner 1988). Because of their high level of precision, detailed process models usually are not transferable from one organization to another (Sommerville 2001, p. 563).

If a complex phenomenon, like software process, is to be represented in detail the resulting model becomes very complex (Humphrey 1989, p. 249) and creation of several different models at different levels of abstraction may be needed (Sommerville 2001, p. 563). High-level process models through their lower level of detail reduce the difficulty associated with understanding a complex phenomenon. Thus, even when ambitious objectives, such as detailed software process definition, are to be achieved, simplistic high-level process models serve as a useful starting point. The common high-level process models are discussed in more detail in the following chapter.

3 HIGH-LEVEL PROCESS MODELS

This chapter discusses the common high-level process models (life cycle models) for software development. As was already proposed above, defining a process model for each software development project is important. Also, the fact that process models can be described at different levels of detail was illustrated. Further, the usefulness of simplistic high-level process models as a starting point for a detailed software process definition was underlined.

It is important to understand that the process models presented below are of a generic nature and are not to be considered as definitive descriptions of a software process. Instead, they should be treated as useful abstractions, which can be used to illustrate the differences between the different approaches to software development. Different approaches are needed when different types of software, or different parts of a software product are developed by different organizations in different circumstances. It must also be noted that the approaches presented here are not mutually exclusive, but can be used as combinations.

3.1 Code-and-fix

This model, also known as build-and-fix, is seldom useful but nonetheless common (McConnell 1996, p. 140). If the organization has not chosen a process model, this model is probably the default model used (McConnell 1996, p. 140). According to this model, the organization starts with a general idea of what to build and then proceeds to build the full system with minimal or no specifications (Boehm, Gray et al. 1984).

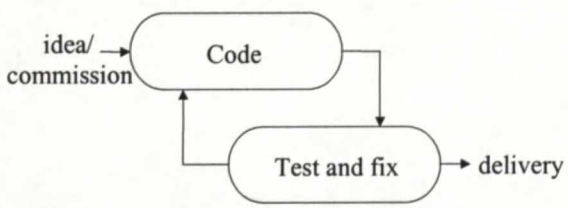


Figure 3. The software process using code-and-fix (based on Boehm 1988).

Code-and-fix is the basic process model used in the earliest days of software development (Boehm 1988) and can be considered to contain two steps: 1) write some code, 2) test and fix the problem in the code (Boehm 1988). In other words, some coding is done first and the questions related to requirements, design, test, and maintenance are dealt with later (Boehm

1988). The resulting product is reworked if necessary until it satisfies the customer (Boehm, Gray et al. 1984).

This model has two advantages. It is easy to use and does not require the development organization to spend any time on planning, documentation, quality assurance, or standards enforcement, that is, on any activities other than pure coding (McConnell 1996, p. 140). Signs of progress in form of software are achieved immediately. First of the primary difficulties, in turn, is that after a number of fixes the code becomes so poorly structured that subsequent fixes are very expensive (Boehm 1988). This makes the model unsuitable for projects of any reasonable size (Boehm, Gray et al. 1984) and underscores the need for design phase prior to coding (Boehm 1988). Second difficulty is that even if the resulting software product is well-designed, it may easily match the user's needs poorly and thus, become either rejected outright or expensively redeveloped (Boehm 1988). This difficulty makes the need for requirements specification phase prior to design evident (Boehm 1988). Third problem associated with this model is that the resulting code is expensive to fix and maintain because of poor preparation for testing, modification (Boehm 1988). Thus, it is clear that explicit recognition of testing phase, as well as test and evolution planning early in the software process are needed (Boehm 1988). Finally, code-and-fix doesn't provide any means of assessing real progress, or managing risks and quality (McConnell 1996, pp. 140, 156).

3.2 Waterfall model

As early as 1956 experience with large software systems had led to the recognition of the problems associated with the code-and-fix model and to the development of a sequential software development process approach to address them (Boehm 1988). This sequential approach was derived from other engineering processes and is the oldest (Sommerville 2001, p. 45) and the most widely used (Pressman 2000, p. 29) of the published process models. It first appeared in the paper of Benington (1983, first published 1956) and became later known as the waterfall model, also known as the linear or linear sequential model.

The waterfall model proposes a systematic, sequential approach to software development (Pressman 2000, p. 26) and represents the fundamental software development steps as discrete successive phases (Sommerville 2001, p. 44). Each phase requires well-defined input, and results in well-defined products (Fairley 1985, p. 37), that is, in principle, one or more

documents which are approved (Sommerville 2001, pp. 45-46). The products of the previous phase cascade to the next, thus acting as an input for that phase. The software product is not delivered until the whole linear sequence has been completed (Pressman 2000, p. 33).

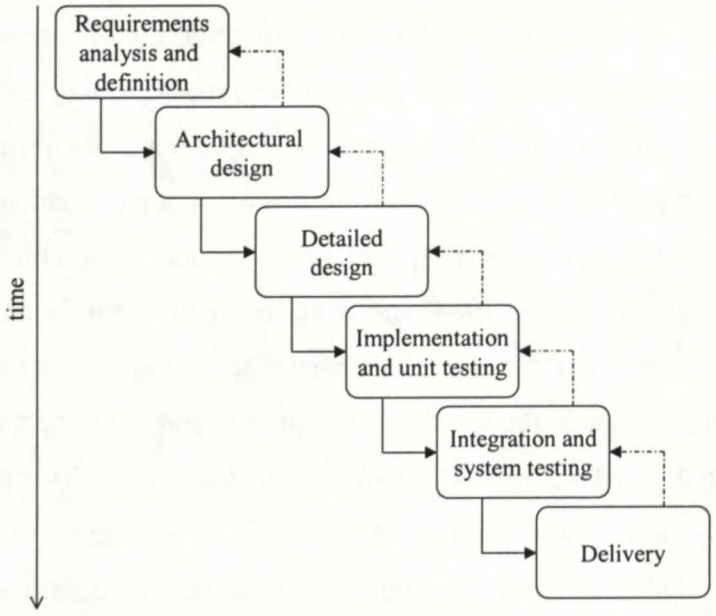


Figure 4. The waterfall with backward loops (McConnell 1996, p. 137; Sommerville 2001, p. 45).

In the pure waterfall model the phases are discontinuous (McConnell 1996, p. 136) – the previous phase should have finished before the following phase starts (Sommerville 2001, p. 46) and a review is held at the end of each phase to determine whether the development project is ready to advance to the following phase (McConnell 1996, p. 136). The central idea behind this approach is to eliminate requirements and design errors from a software product before implementation begins (Fairley 1985, p. 41). Despite this, however, problems with, for example, design sometimes become revealed during coding, thereby leading to an iteration of the development activities (Sommerville 2001, p. 46). Consequently, a real software process often involves overlapping, interaction and feeding of information among the phases (Fairley 1985, p. 41; Sommerville 2001, p. 46) and thus, can not be represented with a strictly linear model (Sommerville 2001, p. 46).

The above problem associated with the strictly linear approach was identified in the Royce's (1987 first published 1970) highly influential refinement (Boehm 1988), which proposed an addition of backward loops to explicitly accommodate iteration and feedback in the software process (Boehm 1987). Unfortunately, in the model, where documentation plays an essential

role, the iterations involve reproducing and reapproving documents and hence, are costly, and involve significant rework (Sommerville 2001, p. 46). Accordingly, Royce (1987) suggested that iterations should be confined as much as possible to neighboring phases in order keep the scope of resulting rework in manageable limits. Moreover, he proposed a prototyping step (see section 3.3.1) running in parallel with design phase in order to ensure that high-risk issues are well-understood and major changes and rework later in the process avoided. In practice, a quite common way to reduce rework is to freeze parts of the software process, such as the requirements specification, and to continue development with the later phases after a small number of iterations (Sommerville 2001, p. 46). Possible problems resulting from the earlier, now frozen phases, but occurring during the later phases are either ignored, left for later resolution, or programmed around (Sommerville 2001, p. 46). These practices, however, easily lead to badly structured software systems (Sommerville 2001, p. 46). A premature freezing of requirements, in turn, may result in a product that doesn't meet the needs of a customer (Sommerville 2001, p. 46).

One modification of the waterfall model is waterfall with overlapping phases. This model suggests, that some of the problems of the linear model can be corrected, if the process phases are allowed to overlap and the emphasis on documentation is reduced. The model recognizes that important insights concerning the early phases of a software process are sometimes gained when working with the later phases has already begun and proposes that the phases should overlap. The idea here is to provide the development personnel with continuity between the phases so that the excessive documentation becomes unnecessary. However, this does not come without cost. Overlapping results in performing development activities in parallel, which may lead to miscommunication and inefficiency. There is also the problem of ambiguous milestones, which results in difficulties when trying to track the project progress accurately. (McConnell 1996, pp. 143-145).

Still another modification of the waterfall model, waterfall with subprojects, addresses the problem of completely finishing previous phase before beginning with the next phase. In this model the software system architecture is broken into logically independent subsystems. The central idea is not to delay implementation of those parts of a software system that are easy to design just because the design of difficult parts is not ready. The subsystems are tackled in separate projects each of which can proceed at their own pace. The main risk associated with this particular modification is unforeseen interdependencies between the subsystems. These

interdependencies should, naturally, be taken into account when dividing the system development into separate subprojects. (McConnell 1996, p. 145)

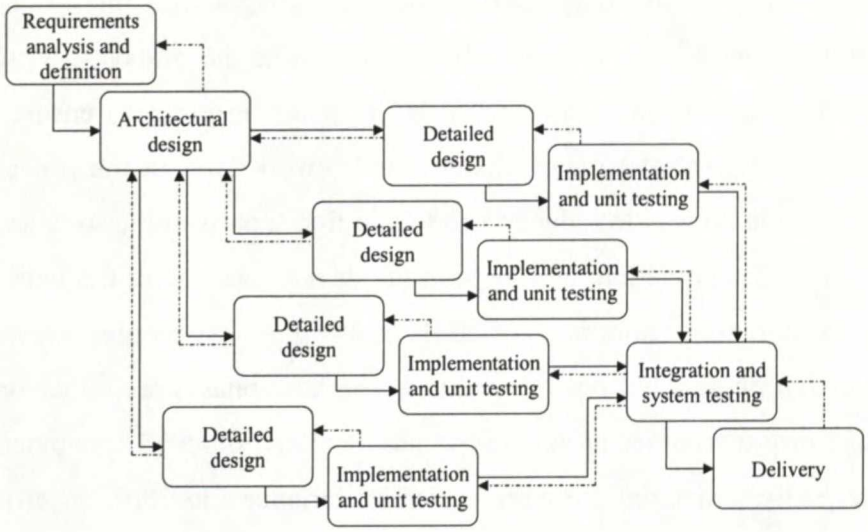


Figure 5. The waterfall with subprojects (McConnell 1996, p. 145).

As for the difficulties associated with the waterfall model and its modifications, the waterfall model's inherent assumption that a product will not be delivered until the whole linear sequence is completed (Pressman 2000, p. 33), is problematic from the customer's point of view, as tangible signs of progress, especially in form of software, are not generated until the very end of the development project (McConnell 1996, p. 139). Moreover, the waterfall model expects customers to be able to fully define and commit to a set of requirements at an early stage of the software process (Sommerville 2001, p. 46). This is difficult, if not impossible, for the customer (Brooks 1987). As the development project proceeds and time passes, the requirements both change and become more detailed, thereby resulting in a constant pressure for specification change. However, the requirements specification frozen at the outset of the software process does not allow responding to these evolving customer requirements (Sommerville 2001, p. 46). Also the errors and omissions in the original software requirements are not discovered until the software product is finally put into use (Sommerville 2001, p. 46). In order to meet the new requirements and correct the discovered errors, a costly repetition of some or possibly all of the previous software process phases is now needed (Sommerville 2001, p. 46). Apart from this, the waterfall model's emphasis on fully elaborated documents as a completion criteria for early requirements and design phases has sometimes resulted in writing elaborate specifications of poorly understood requirements, followed by the design and development of large quantities of unusable code (Boehm 1988).

In sum, the natural uncertainty existing at the beginning of many software development projects can not be easily handled with the waterfall model (Pressman 2000, p. 29).

On the other hand, the waterfall model is a simple management model (Sommerville 2001, p. 51), which helps to manage software development's inherent complexity by providing requirements stability (McConnell 1996, p. 136). Also, the separation of design and implementation, and thus the lack of continuous changes, which are a huge and common source of potential errors (McConnell 1996, p. 137), should lead to reliable systems (McConnell 1996, p. 156) making this model suitable for situations where quality requirements dominate cost and schedule requirements (McConnell 1996, p. 137). Also, planning overhead is minimized as all the planning is done up front and wasted effort is minimized due to structure provided by the model (McConnell 1996, pp. 136-137). As to tracking project progress, the lack of tangible results in the form of software until the end of process is compensated by the provision of milestones with documentation, which provide managers indications of the project status throughout the process (McConnell 1996, p. 136).

In summary, the waterfall model is a valid model of the software process in situations where the product definition is stable, and product requirements (Pressman 2000, p. 28; Sommerville 2001, p. 46), technical methodologies, and product architecture are well-understood (McConnell 1996, pp. 136, 156). It performs well when developers have previously developed similar systems, and it is therefore possible to write a reasonably complete set of specifications for the software product already at the beginning of the development project (Fairley 1985, p. 41). In well-understood but complex projects this model helps the development team to tackle complexity in an orderly way (McConnell 1996, p. 137). Despite its problems the waterfall model reflects engineering practice and consequently, is still widely used for the software development (Pressman 2000, p. 29; Sommerville 2001, p. 46).

3.3 Iterative process models

The essence of iterative process models is an evolving software specification that is developed in conjunction with the software (Sommerville 2001, p. 51). In other words, as the waterfall model conceives the software process as a manufacturing process based on a stable product definition prepared early in the process, the iterative models are based on the idea of accommodating uncertainty and consider the software development as a learning process.

However, unlike the unsystematic code-and-fix model (which, in a sense, is iterative), iterative process models, as referred to here, are about systematic, intentional iteration.

Need for iterations, that is, repeating parts of the process, arises when modifications and corrections to the products of the previous phases of the software process are proposed. These might be due to inaccuracies, and incompleteness in those products or changes in customer requirements (Fairley 1985, p. 47). Requirements changes, in turn, may result from unknown requirements, unstable requirements, or misunderstandings between developers and users (Humphrey 1989, pp. 255-256). Apart from these, also changes in schedules, priorities, and budget dictate modifications (Fairley 1985, p. 47).

The need for iteration in the software process arises for the most large software systems (Sommerville 2001, p. 51). Developing complex, large software products in a one long, linear sequence (waterfall) is extremely difficult because this kind of project easily exceeds our human intellectual capabilities for management, planning, and control (Mills 1976; Humphrey 1989, p. 62). The iterative models tackle this problem by partitioning the process into smaller, easier to handle iterative steps, and not trying to do everything right at once.

Process models based on prototyping, increments, and spiral development have been proposed to specifically accommodate iteration in the software process. Although iterative, also these models require that the changes are, from time to time, temporarily frozen while the development proceeds (Humphrey 1989, p. 255). A continuous change in, for example, interfaces would make designing, building, and testing a program impossible (Humphrey 1989, p. 255). Moreover, the change must be controlled in order to prevent the software process from becoming unstable and to ensure productivity and quality (Humphrey 1989, p. 255). Also the project progress monitoring must be arranged for, because producing deliverables in the same way than when using the sequential waterfall approach isn't now cost-effective. The exemplary ways of improving progress visibility are establishment of milestones, standardized documents, and management sign-offs (Fairley 1985, p. 42).

3.3.1 Prototyping-based process models

The central idea in prototyping is to develop a preliminary implementation, a prototype, of the software product to be built, expose it to evaluation in order to elicit comments, determine

feasibility, or investigate technical or some other issues, and modify the prototype through many versions until it is adequately tuned to satisfy the needs (IEEE Std 610.12 1990; Sommerville 2001, p. 46). The feedback in form of, for example, customer comments leads to iteration of development activities, thereby providing both the developer and the customer with a better understanding of the product to be developed (Pressman 2000, p. 29). Consequently, prototyping is an ideal approach to a software development in situations where the requirements are changing rapidly, the application area is not well-understood (McConnell 1996, p. 147), defining a software product without some exploratory development is not possible, or a better understanding of the customer's requirements needs to be gained (Fairley 1985, pp. 49-50).

The prototyping process begins with the developer and the customer together defining the overall objectives for the software product to be built (Pressman 2000, p. 29). After this, a prototype of the product is rapidly developed from these abstract specifications (Sommerville 2001, p. 44). In order to enable rapid development of the prototype, the fundamental software development steps of requirements specification et cetera are carried out concurrently rather than sequentially (Sommerville 2001, p. 46). This requires a rapid feedback across these steps (Sommerville 2001, p. 46). The speed of development is essential for this approach to be effective, as it is important that the customer can assess the results, experiment with them, and recommend changes early in the process (Andriole 1994). Moreover, the rapid development helps control development costs (Sommerville 2001, p. 172).

Expressing the real requirements is usually difficult for customers (Sommerville 2001, p. 172). In fact, it has been argued that requirements can never be stated fully in advance, because the development project itself changes the customer's perceptions of what is possible and desirable (McCracken and Jackson 1982). This problem can be alleviated with a careful requirements analysis and systematic reviews of the proposed requirements (Sommerville 2001, p. 172). Prototyping supports these activities by giving the customer a feel for the actual system. Experimenting with a prototype helps the customer identify strengths and weaknesses in software and thus, propose new, previously unidentified requirements (Sommerville 2001, p. 172). Apart from this, the experimentation may reveal inconsistencies, omissions, or errors in the proposed requirements, a significant risk in software development, and thus aid requirements validation and risk reduction (Boehm, Gray et al. 1984; Sommerville 2001, p. 172). This, in turn, helps reduce overall development costs, as fixing requirements errors and

omissions later in the software process can be very costly (Boehm 1976). Further, the iterative experimentation and evaluation, and thus, incremental development of the requirements specification mean, that the customer can try out the requirements before agreeing to them. This, naturally, reduces the customer's uncertainty about the nature of the software system to be built (Sommerville 2001, p. 172). Finally, the experimentation with the product brings the misunderstandings between developer and customer to light (Gordon and Bieman 1995) and thus increases the probability of project success (Bersoff 1991). All this should help achieving a product which more closely matches the needs of the user (Gordon and Bieman 1995).

As to the process of prototyping, it is paramount that the objectives of prototyping are made explicit from the beginning (Humphrey 1989, p. 275; Gordon and Bieman 1995). It is especially important to explicitly choose between two different prototyping approaches, that is, evolutionary and throw-away prototyping (Bersoff 1991). In evolutionary prototyping the prototype is refined according to user comments until it is adequate and ready to be delivered to the customer as a final software system (Sommerville 2001, pp. 174-175). Throw-away prototyping, on the other hand, is an approach where the prototype does not evolve into a final product, but is discarded after it has served its purpose which might be, for example, to aid requirements engineering (Pressman 2000, p. 283). Defining an objective helps the customer understand what to expect from the prototype (Sommerville 2001, p. 173) and also prevents waste of resources for building quality into something to be discarded (Bersoff 1991).

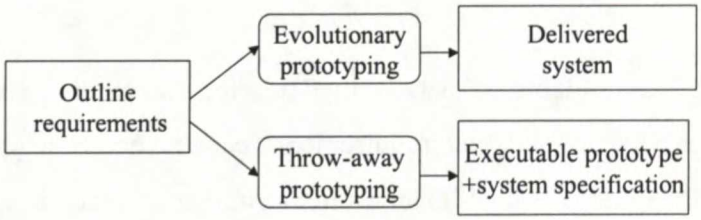


Figure 6. The two prototyping approaches (Sommerville 2001, p. 175).

It is usual, that compared to a final software product the prototype (early versions of a evolutionary prototype and all the versions of a throw-away prototype) exhibits limited functional capabilities, lower reliability, and/or inefficient performance (Fairley 1985, p. 49). This enables the reduction of prototyping costs and also helps to deliver an initial version of the product more quickly (Sommerville 2001, p. 174). Consequently, a decision must be made concerning what is left out and what is implemented in the prototype (Sommerville 2001, p.

173). After the prototype is constructed it is evaluated by the customer, and thereby used to refine the requirements for the software to be developed (Pressman 2000, p. 29). For the evaluation to be successful, it is important that the customer is committed to it and that he also is capable of giving feedback and making requirements decisions in a timely fashion (McConnell 1996, p. 437; Pressman 2000, p. 284).

The nature of the software product to be built affects the nature and extent of prototyping (Fairley 1985, p. 50). New versions of an existing product can often be developed with little or no prototyping (Fairley 1985, p. 50). A totally new software product, in turn, may be developed by iterating through series of successive designs (evolutionary prototyping) and implementations or at least some prototyping is involved during the analysis (throw-away prototyping) (Fairley 1985, p. 50; Royce 1987). The following two sub-sections discuss the two prototyping approaches in more detail.

Evolutionary prototyping

The approach called evolutionary prototyping is often cited to have been briefly established at the end of (McCracken and Jackson 1982). In evolutionary prototyping the customer is provided with an initial, relatively simple implementation of the software product which is then gradually modified and augmented according to the customer evaluation until the software system ready to be delivered to the customer as a final product has been developed (Sommerville 2001, p. 175).

When this approach is used, the development usually begins with the parts and aspects of the software product which are best understood, and have the highest priority (Sommerville 2001, p. 175). The requirements that are of lower priority or vague are implemented when and if the customer demands them (Sommerville 2001, p. 175). The prototype evolves and is developed further based on the received customer feedback until it ultimately has become the software system which is required (Sommerville 2001, p. 174). As the prototypes in this approach evolve into a final system, they should be developed to the same organizational quality standards as any other software (Sommerville 2001 p. 175). Thus, for example reliability and efficiency requirements can not be relaxed to the same level as in throw-away prototyping, where the prototype doesn't evolve into a final system (Sommerville 2001, p. 175).

In addition to the typical advantages associated with prototyping approach (better correspondence with customer requirements et cetera discussed above) one advantage of evolutionary prototyping is the reduction in the development effort of a software system (Gordon and Bieman 1995). The evolutionary prototyping also addresses risks early, as the high-priority areas of the software product are developed first (McConnell 1996, p. 434). The approach also gives the developer early feedback concerning whether the final product will be acceptable (McConnell 1996, p. 440).

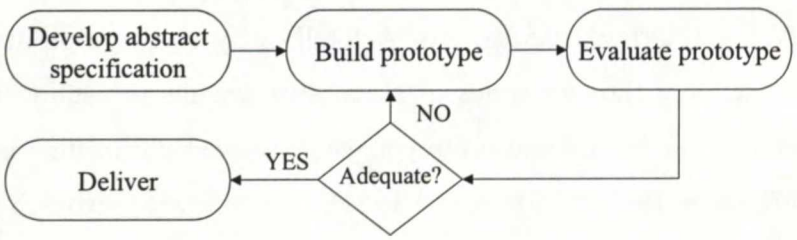


Figure 7. The process of evolutionary prototyping (Sommerville 2001, p. 176).

As to the problems associated with evolutionary prototyping, producing documentation that reflects every new version of the software is not cost-effective when new versions are developed at very short intervals (Sommerville 2001, p. 47). The lack of regular deliverables is problematic from the management's point of view, as deliverables are needed to measure the project progress (Sommerville 2001, p. 47). However, from the customer's point of view, the progress visibility in prototyping, which produces steady visible signs of progress, is better than for example in waterfall approach, where deliverables in the form of software are not delivered until the end of the development project (McConnell 1996, pp. 147, 433).

Further, continual changes, typical of evolutionary approach, may corrupt the software structure and thus, result in difficulties (Gordon and Bieman 1995) and increased costs when trying to incorporate further changes. The poor structure, in turn, leads to problems in software maintenance, as the software structure becomes complicated and difficult to understand (Gordon and Bieman 1995; Sommerville 2001, p. 51). These problems resulting from the poor structure get even worse when combined with the lack of detailed documentation (Boehm, Gray et al. 1984). The structural quality suffers also if discarded design alternatives are, for some reason, left unremoved from the final product (Gordon and Bieman 1995). As a result of all these problems, software systems developed using evolutionary prototyping tend to have a fairly short lifetime (Sommerville 2001, p. 178).

In addition to problems with maintenance, some additional problems are due to the fact that in evolutionary prototyping there is no detailed specification and often, there also may not even be a formal requirements document (Sommerville 2001, p. 174). In prototyping the specification is the result of a prototyping process (McConnell 1996, p. 147). However, as was discussed above, producing documentation that reflects every new version of the software in detail is not cost-effective. This lack of a detailed specification leads into problems during verification, where the software product's conformity with its specification is evaluated (Sommerville 2001, p. 177). Also, as there is no explicit statement of purpose, the validation of the product's suitability for its intended purpose is difficult (Sommerville 2001, p. 177). Consequently, in case of evolutionary prototyping the validation and verification of a software product can only evaluate if the product is adequate enough for its intended purpose (Sommerville 2001, p. 177). This is problematic, as only subjective judgments concerning adequacy can be made (Sommerville 2001, p. 177).

The lack of the detailed specification leads also to contractual problems as the contract between the developer and the customer can not be based on a specification as usual (Sommerville 2001, p. 177). One solution would be fixed price contracts, but these are not liked by developers, because controlling changes requested by customers would become difficult (Sommerville 2001, p. 177). The contract could, of course, be based on the time spent on the project, but now the customer is not likely to accept this (Sommerville 2001, p. 177). This is because it is impossible to know at the beginning of the project how long or how many iterations the development of an acceptable product takes (McConnell 1996, p. 147).

The uncertainty about the duration of a development project, discussed above, is not problematic only from the customer's point of view. The uncertainty about the project duration, especially together with the lack of documentation, leads to a diminished project control (McConnell 1996, p. 436). Apart from this, it may also result in unrealistic schedule, budget, and performance expectations (McConnell 1996, p. 442). This makes expectations management important when evolutionary prototyping is applied (McConnell 1996, p. 442).

Finally, when deciding whether to use evolutionary prototyping it is necessary to determine whether the software product to be built is amenable to prototyping (Pressman 2000, p. 284). Prototyping is a valid approach for small software systems (less than 100,000 lines of code) or medium sized systems (up to 500,000 lines of code) especially if these systems have a fairly

short lifetime (Sommerville 2001, p. 47). However, large and long-life systems are likely to be too complex for evolutionary prototyping (Pressman 2000, p. 284), although, also positive experiences have been reported (Gordon and Bieman 1995). If it is possible to partition the complexity typical of large software systems, portions of the system might be amenable to prototyping (Pressman 2000, p. 284). However, a better solution would probably be a mixed process model that incorporates the best features of waterfall and prototyping approaches (Boehm, Gray et al. 1984). Prototyping approaches (both throw-away and evolutionary) can be used to resolve the possible uncertainties, and the well-understood parts of the product can be developed according to the waterfall approach (Boehm, Gray et al. 1984).

Throw-away prototyping

Compared to evolutionary prototyping, where the goal is to evolve the prototype into a final product to be delivered to a customer, the objective of throw-away prototyping is to help refine and clarify requirements and design specifications (Bersoff 1991; Sommerville 2001, pp. 46, 175). Consequently, in throw-away prototyping the problems associated with the lack of specification typical of evolutionary prototyping are not acute. In addition to requirements clarification, the throw-away prototype can be used to evaluate the feasibility of the proposed solution, investigate technical issues, or provide additional information for risk assessment purposes (Gordon and Bieman 1995; Sommerville 2001, pp. 46, 179).

In throw-away prototyping a deliberate decision is made at the outset of the project to discard the prototype after the requirements have been gathered and the specification is complete or some other purpose has been served. As the throw-away prototype does not evolve into a final product the problems of short lifetime and poor long-term maintainability due to continuous changes are not really a problem in throw-away prototyping. Although the prototype must be amenable to rapid changes during development, long-term maintainability is not required. (Sommerville 2001, p. 174-175).

The generic process of throw-away prototyping is presented below. First, an outline of requirements is developed and then, a prototype is developed from this specification. Compared to evolutionary prototyping where the best-understood requirements are implemented first, requirements analysis in throw-away prototyping begins with the requirements which are poorly understood (Sommerville 2001, p. 175). In fact, clearly understood requirements may never become prototyped.

The prototype is exposed to a user for experimentation and refined according to the resulting customer feedback. This iterative process of evaluation and refinement is repeated until the customer is satisfied with the prototype. The waterfall or some other process model is now entered and the specification for the final product is derived from the throw-away prototype. Thus, throw-away prototyping acts as a springboard for other process models. After the prototype has served its purpose, it is discarded and the software product is re-implemented in a final version. Some of the prototype components may be reused in the final product and thereby both the development cost and time may be reduced. After the validation has shown that the software product is good enough for its intended purpose, the product is delivered to the customer. (Sommerville 2001, p. 179)

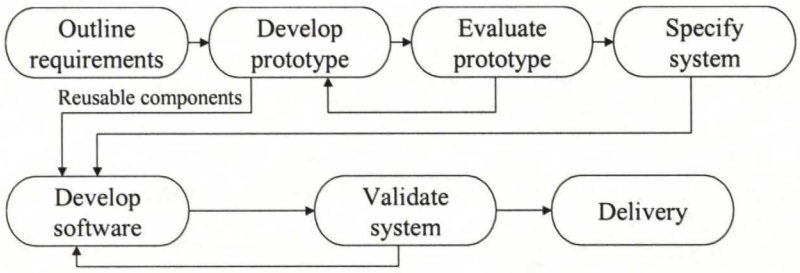


Figure 8. The software process using throw-away prototyping (Sommerville 2001, p. 179).

As to speeding up the development of the prototype by relaxing quality standards and performance criteria, this can be done to a level which is not possible when a prototype will evolve to a final product, in other words, evolutionary prototyping is applied. Also the development language will often be different from the implementation language of the final software system. Moreover, throw-away prototypes do not have to be executable and working software prototypes. For example, paper-based mock-ups have been effectively used to represent the software system's user interface during the requirements engineering. (Sommerville 2001, pp. 179-180).

Also throw-away prototyping has its disadvantages. First of all, it may be difficult for a customer to understand the difference between a limited prototype and an actual, complete, software product. Customer might be unaware of the fact that for the prototyping purposes the quality and maintainability requirements may have been relaxed and that the product really has to be rebuilt (Pressman 2000, p. 30). He might therefore demand that a rough prototype is quickly converted into a final product by applying only few fixes (Pressman 2000, p. 31). However, if this is done, quality almost always suffers as a result (Pressman 2000, p. 31). One

reason for this is the fact that quality and performance standards are often relaxed for prototype development (Sommerville 2001, p. 180) and that refining the prototype to meet the performance, security, robustness, and reliability requirements which were relaxed or ignored during the prototype development may even be impossible (Sommerville 2001, p. 180). Moreover, the results of rapid development, such as lack of documentation and degraded structure, would lead to problems with long-term maintenance (Sommerville 2001, p. 180). Brooks (1982, p. 116) has gone as far as stating that it is nearly impossible to get it right the first time and thus, we should always plan to throw one away. Further, he states, that delivering the throw-away prototype buys time but does so only at the cost of agony for the user, and a bad reputation for the product. Thus, it is important that the customer and developer together define the objectives of prototyping at the beginning and agree whether the prototype is built merely for modeling purposes (throw-away prototyping) or extended to a final product (evolutionary prototyping) (Pressman 2000, p. 31).

A typical problem associated with executable throw-away prototypes is that the mode of use of a prototype may not correspond with the way the final system is used (Sommerville 2001, p. 180). Also the evaluators and testers of a prototype may not be typical of the users of the final system (Sommerville 2001, p. 180). Further, the evaluators may adjust their behavior according to a prototype's defects and thus, avoid using prototype features which are, for example, annoying or inconvenient (Sommerville 2001, p. 180). All these problems result in the prototype evaluation that does not accurately reflect the use of the final system and thus, the prototype's value as a tool for specification clarification diminishes.

From the management's perspective there is also a risk of inefficient use of prototyping time (McConnell 1996, p. 440). Developers might, for example, provide the prototype with unnecessary extra features (McConnell 1996, p. 440). This may result in needless delays in the development project and also increase the development costs. Thus, prototyping projects need to be managed carefully and controlled with regard the features to be implemented in the prototype (McConnell 1996, p. 440). For small software systems evolutionary prototyping is preferable to throw-away prototyping because the overhead of creating a throw-away prototype makes it economically unfeasible (Gordon and Bieman 1995).

3.3.2 Increment-based process models

According to increment-based process models, the software product to be developed is decomposed into small subsets that are progressively developed, installed, used, evaluated, and then enhanced (Humphrey 1989, p. 255). This approach makes an early delivery of an operational subset of the product to the customer possible. Quick delivery of a limited version is sometimes required, for example, to meet competitive pressure. This subset is enhanced with later increments to build increasingly more complete versions of the software product.

Delivering the software product in operational increments that are successively integrated to each other requires that the high-level design is well-planned and controlled (Humphrey 1989, p. 256). Whereas the software product in prototyping approaches evolves through relatively uncontrolled change, in incremental models an orderly change is emphasized. Integration and modifiability must be considered from the beginning (Boehm 1976; Mills 1976). Moreover, the functionality provided by the increments must be phased to meet end user needs and each software component design must synchronize with these needs (Humphrey 1989, p. 62).

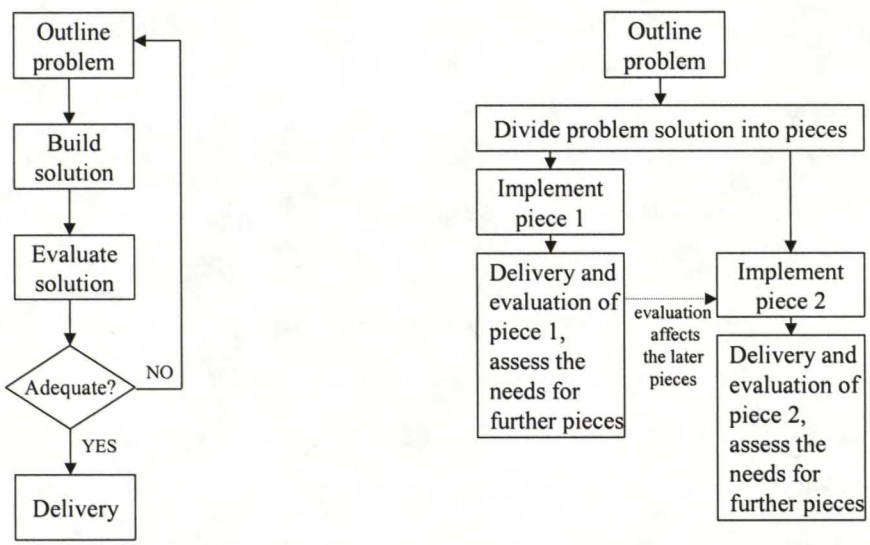


Figure 9. Whereas models like evolutionary prototyping iterate with the entire solution, increment-based models divide the solution into pieces and experiment and iterate with them.

Interestingly, iterative and incremental approaches are sometimes considered to be two different, albeit not mutually exclusive, approaches. The iterative approach is then considered as an approach where the system or part of it is reworked and the incremental approach, on the other hand, is seen as the dividing of a development project into multiple pieces. However, in this study, increment-based process models are discussed under iterative process

models. This is because iteration is not here seen only as the repetition of the software process steps to rework the product developed so far, but also to develop whole new subsets for the product. This latter form of iteration is present in increment-based process models where each iteration round consists of repeating certain development steps to build new product subsets. Moreover, in increment-based models the repetition of development tasks is due to both division of the development project into multiple pieces and the gradual requirements analysis. Incremental models accept that new requirements emerge and old requirements become more detailed as the development project proceeds and when the subsets of the software product are released and experimented with. This evolving nature of requirements incremental models accommodate through repetition of software process steps to develop new subset and fix the old ones. Hence, also in incremental models, the software specification is developed in conjunction with the software.

Examples of increment-based process models are incremental development, iterative enhancement, and evolutionary delivery. These are discussed in the following three sub-sections.

Incremental development

Incremental development model proposed, for example, by (Dyer 1980; Linger 1980; O'Neill 1980; Quinnan 1980; Mills 1999) was introduced as a development approach which would reduce rework in a software process and at the same time give a customer an opportunity to delay some decisions on the detailed requirements until some experience with the system had been gained (Sommerville 2001, p. 51). In incremental development software is developed in small but usable pieces that can be delivered to a customer (Pressman 2000, p. 34). Each increment is an operative subset of the software system to be produced and builds on the increments that have already been developed (Pressman 2000, p. 34). As to the delivery of the increments, it is important to notice that incremental development does not necessarily imply incremental delivery to a customer. That is, incremental development can be used as a purely internal development approach without delivering increments to a customer on one-by-one basis (McConnell 1996, p. 553).

When incremental development is used, customers identify how important the services to be provided by the software product are to them (Sommerville 2001, p. 52). After this, the

number of increments to be delivered and the functionality to be delivered with each increment is defined (Sommerville 2001, p. 52). However, only the services to be delivered in the first increment are at this phase defined in detail (Sommerville 2001, p. 52). Usually, the first increment is a kind of a core product which then is augmented with supplementary features delivered through later increments (Pressman 2000, p. 34) so that the system functionality gradually improves with each increment (Sommerville 2001, p. 52). The most important functionality is delivered through the early increments (Sommerville 2001, p. 52).

In incremental development, increments are developed in separate stages after the overall system architecture has been defined. Detailed design, coding, and testing (in the below figure, these are under the phase "develop system increment") occur within these separate stages (McConnell 1996, p. 149). The process of development, validation and integration continues until the delivered increments form a complete product.

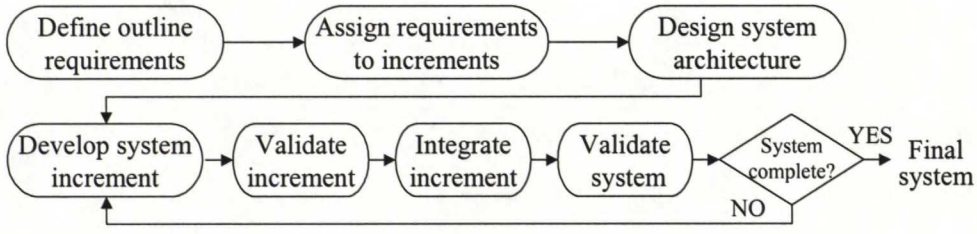


Figure 10. The process of incremental development (the possible delivery of product subsets to a customer not presented in the figure) (based on Sommerville 2001, p. 52).

The partitioning of the total solution into increments should satisfy the following three requirements at minimum (Dyer 1980). First, the resulting partitioning should be natural and logical with respect to the operational system or application. Second, it should result in maximal separation of the increment's functions from functions in other increments. Third, the development of increments should be phased so that modification of previously completed increments due to the implementation of subsequent increments is minimized. Thus, openness of the software architecture is emphasized in the design phase in order to aid the evolution of the software into a progressively more refined software system through the easy integration of increments to each other. Consequently, when architecture is designed a top-to-bottom understanding of the software to be developed is required and many iterations may be required to arrive at a suitable design structure (Linger 1980).

Incremental development can be considered to contain other process models. That is, the phase of "develop system increment" is carried out using the most appropriate approach and thus, for example, a well-specified increment may be developed using the waterfall model. Different increments can be developed using different high-level process models. (Sommerville 2001, p. 52)

After an increment is completed, it can be delivered to a customer who can then put the increment into service and thereby benefit from the functionality provided by it (Sommerville 2001, p. 52), but also use it as a basis for evaluation (Pressman 2000, p. 35). The customer's experimentation with the delivered subsets of the product helps customer clarify requirements for the increments scheduled for later delivery, and for later versions of the increments currently under experimentation (Sommerville 2001, p. 52). However, requirements specification for the increment currently under development is frozen and thus, changes to that particular increment are not accepted (Sommerville 2001, p. 52).

In incremental development some of the problems of constant change typical of, for example, evolutionary prototyping, are avoided (Sommerville 2001, p. 178). An overall system architecture is established early in the process to act as a framework within which the increments are gradually developed and delivered (McConnell 1996, p. 149; Sommerville 2001, p. 178). Once validated and delivered, the framework and components are only changed if errors are discovered (Sommerville 2001, p. 178). Requirements changes concerning validated increments are delivered in the new versions of these increments. Finally, plans and documentation are produced for each increment (Sommerville 2001, p. 178). All this enhances the maintainability of the resulting product.

Incremental development has a number of advantages. First, customers can benefit from the functionality delivered through early increments and thus, they don't have to wait until the product is 100% complete (McConnell 1996, p. 149). Moreover, the first increment satisfies the customer's most critical needs (Sommerville 2001, p. 52). Second, incremental development gives customer an opportunity to delay decision making on detailed requirements until some experience with a system has been gained (Sommerville 2001, p. 52). Third, there is a lower risk of overall project failure, as at least some of the product functionality will be successfully delivered (Sommerville 2001, p. 52). Fourth, as the most important functionality can be delivered first and later increments are integrated with it, the

most critical services receive the most testing (Sommerville 2001, p. 53). Fifth, the delivery from the most important to least important forces customer to prioritize (Humphrey 1989, p. 62; McConnell 1996, p. 553). Sixth, increments allow some user feedback early in the process while at the same time limiting system errors as the developers are not concerned with interactions between quite different parts of the system (Sommerville 2001, p. 178). Interfaces of a delivered increment are frozen and later increments are adapted to and tested against these interfaces (Sommerville 2001, p. 178). Seventh, the approach provides tangible signs of progress (McConnell 1996, p. 149). Eighth, if there are problems in development, they come into light during the development of the early increments (McConnell 1996, p. 551).

However, there are also problems associated with incremental development. First, increments should be relatively small in size (no more than 20,000 lines of code) while at the same time they should deliver some functionality (Sommerville 2001, p. 53). Second, mapping the customer requirements onto increments of the right size may be difficult (Sommerville 2001, p. 53). Third, as detailed requirements are not defined until an increment is to be developed, identification of common system facilities used by the various increments is difficult (Sommerville 2001, p. 53). Fourth, incremental development requires careful planning from both the management (distribution of work between increments, schedule) and technical (dependencies between increments) point of view (McConnell 1996, p. 149).

As an overall system architecture is defined before any of the increments are developed, incremental development is a suitable approach in situations, where software systems to be developed are well-understood (McConnell 1996, p. 554). It is also useful in situations, where development projects are large, system nucleus involves a high risk (Boehm and Belz 1990), or customers want to benefit even from the relatively small portions of the product's functionality (McConnell 1996, p. 554). Naturally, this requires that useful subsets of the software product can be developed independently (McConnell 1996, p. 555). Finally, it is a model to be considered when only limited budget or staff is available (Boehm and Belz 1990), or a project deadline has turned out to be too tight, and a complete implementation by the deadline is impossible (Pressman 2000, p. 35).

Iterative enhancement

Approaches like incremental development, tackle the software development in pieces but at the same time require, that overall system architecture is carefully established before any of the increments are developed. This is, naturally, problematic if the problem and its solution are not that well understood in advance. The model of iterative enhancement was developed as a solution to this problem (Basili and Turner 1975). Compared to incremental development, the iterative enhancement allows more uncertainty in the software process and thus, serves as a more flexible alternative to the incremental development.

In the iterative enhancement the software process begins with a simple initial implementation of a properly chosen subset of the software product to be developed. This subset acts as an initial guess in the process of developing the full product and is gradually enhanced through successive iteration rounds until the product is ready. Iteration rounds may involve extensions, but also modifications of the previous implementation. Through this iterative process the software system under development becomes better understood and consequently, fewer and fewer modifications to the existing implementation need to be made towards the end of the development project. (Basili and Turner 1975)

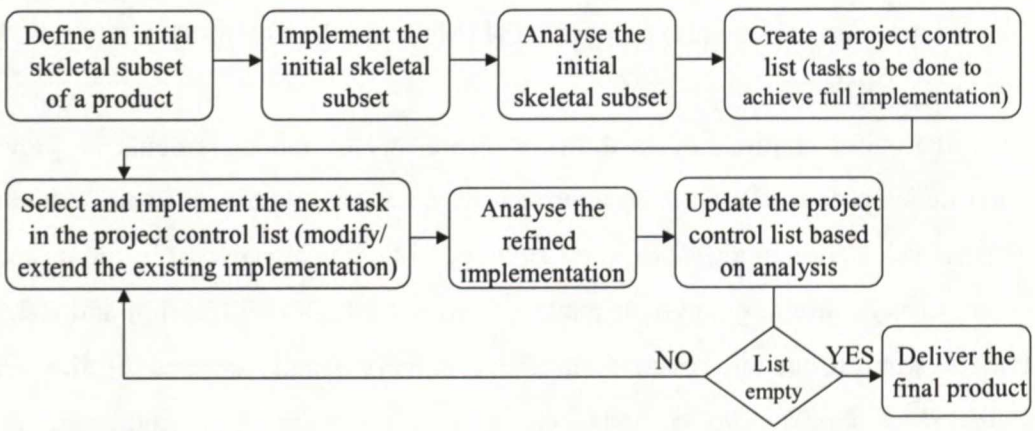


Figure 11. The process of iterative enhancement (the possible delivery of product subsets to a customer not presented in the figure) (based on Basili and Turner 1975).

The product subset first developed should contain the key aspects of the software product under development, and be simple enough to understand and implement easily so that a useful and usable partial product can be delivered to the customer early in the process. Moreover, the subset should be straightforward in overall design, and modular at lower

levels of design and coding, so that it can be easily modified in the iterations leading to the final implementation. (Basili and Turner 1975)

The iterative enhancement involves the creation of a project control list that guides the iterative software process by keeping track of all the tasks that need to be performed in order to achieve the full software product. The tasks in the list include measures that need to be taken to fix flaws discovered in the existing implementation, to design and implement the new features, and to solve the currently unsolved problems. The model emphasizes the importance of conceptually simple tasks that can be fully understood. The progress in software process is achieved in small increments in order to minimize the chance of error in the design and implementation phases. (Basili and Turner 1975)

Each iteration round involves selecting and removing the next task from the project control list, designing implementation for the selected task, coding and debugging the implementation of the task, performing analysis of the existing partial implementation, and updating the project control list as a result of this analysis. Hence, the project control list acts as a measure indicating the distance between the current implementation and the final software product. When the project control list is empty there no more is gap between the existing and final implementation and thus, the iteration process can be discontinued. The project control lists developed and modified during the development project correspond to the partial implementations developed while proceeding towards the final product and thus, are a valuable component of the historical documentation of the project. (Basili and Turner 1975)

In the model of iterative enhancement an essential component is the analysis performed on each successive implementation and leads to the project control list becoming revised. During the analysis multiple items such as the structure, modularity, modifiability, usability, reliability, and efficiency of the existing implementation as well as an assessment of the achievement of the project goals are taken into consideration. User feedback plays an essential role in finding deficiencies in the existing implementation. (Basili and Turner 1975)

The model of iterative enhancement somewhat resembles evolutionary prototyping, where the customer experiments with an initial version of the software product which then is gradually modified to a final working product. However, as the software product developed through evolutionary prototyping may typically suffer from low maintainability and difficulties in

modifying the software, the particular objective of the iterative enhancement is a software product which is modifiable (Basili and Turner 1975). In the iterative enhancement, the progress through iterations is based on simple, well-understood extensions and modifications which, in turn, are based on a better understanding of the problem obtained through the process (Basili and Turner 1975). Moreover, any difficulty in designing, coding, or debugging a modification is considered as a sign of the need to revise the existing implementation (Basili and Turner 1975). Thus, unlike evolutionary prototyping, the iterative enhancement where the design and code are repeatedly examined and reevaluated should lead to a product which is well-understood, and consequently, reliable, modifiable, and maintainable. Finally, in iterative enhancement the importance of the usefulness and usability of the intermediary products, and thereby, possibility to deliver the partial product to a customer is implied.

The advantages and disadvantages of the iterative enhancement model can be derived from the ones of incremental development model, but only on condition that the characteristic features of the iterative enhancement model discussed above are taken into consideration.

Evolutionary delivery

Another variation of incremental approach to software development is evolutionary delivery presented in, for example, (Gilb 1985). Like in the iterative enhancement, the central point in the evolutionary delivery is to start with a basic design which is easy to modify both in long and short terms and use this partial solution in order to learn more about the desired product (Gilb 1988, p. 92). The message of both the iterative enhancement and the evolutionary delivery is that the solution to a problem does not have to be analyzed in detail beforehand (Gilb 1988, p. 92). The problem can be tackled in small pieces, where every piece results in better understanding of the problem and thereby, of the solution (Gilb 1988, p. 92). Thus, both these models serve as more learning-oriented and flexible alternatives to incremental development discussed earlier.

The model of evolutionary delivery, known also as EVO model, is based on the idea of delivering the software product to a customer in very small increments and concentrating on long-term performance and adaptability of the system (Gilb 1988, p. 231). In addition to taking very small steps, the emphasis is on choosing the steps with the highest user value relative to development costs for the earliest delivery (Gilb 1988, p. 90). The EVO software

process is objective driven and both software design and project objectives are based on feedback generated through experimentation with the partial product in a real environment (Gilb 1985). The iteration is guided by clear and measurable functional, quality, and resource objectives which are vital to both the long and the short-term survival of the product under development (Gilb 1988, p. 89). Concentrating on objectives which are less likely to be subject to change than requirements helps prevent the software product from evolving into one that the user does not want or need (Gladden 1982). Also, if there already is some kind of predecessor for the software product to be developed, this model tries to make a substantial use of it by gradually modifying it towards the desired product rather than building a totally new one from scratch (Gilb 1988, pp. 84, 86). This is different from, for example, the waterfall approach, where the old software system is replaced by the new all at once (Gilb 1988, p. 86).

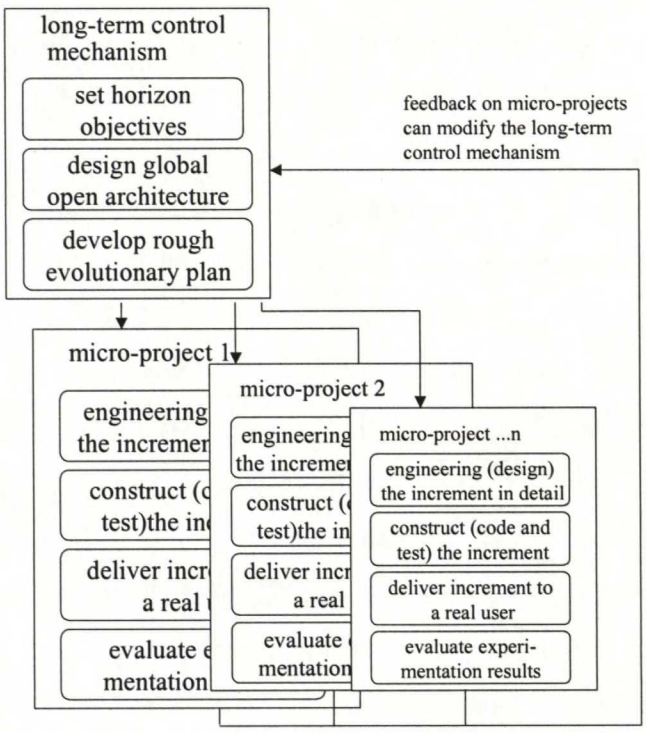


Figure 12. The process of evolutionary delivery (Gilb 1988, p. 88).

The EVO process consists of many micro-projects (Gilb 1988, p. 88). Each micro-project involves the steps of analysis, design, build, and test, and results in an increment that is delivered to a customer and takes the system under development towards the final software product (Gilb 1988, p. 91). The customer feedback resulting from the experimentation with the current implementation is used to refine the existing design and architectural ideas, and to

modify the development project objectives (Gilb 1988, p. 91). Open-endedness of the system architecture is emphasized, for it is needed to enable the easy incorporation of the suggested changes into software product under development, and to achieve a software system which is modifiable and maintainable also in long term (Gilb 1988, pp. 92, 98).

The process of evolutionary delivery uses an adapted form of stepwise refinement where the development steps are defined broadly at first and then further defined with increasing detail (Wirth 1971; IEEE 1990 std 610.12). Thus, the major functional requirements and design specification ideas are first identified, and then distributed to smallest possible but useful delivery steps (Gilb 1988, p. 230). Each micro-project is somewhere between 1% and 5 % of the total project effort in size (Gilb 1988, p. 230). Planning in the evolutionary delivery model concentrates on short-term results - long range plans are made only at a rough level and are adjusted according to experience gained during the implementation of and experimentation with the partial implementations (Gilb 1988, p. 228). Considerable changes in the plans are allowed (Gilb 1988, p. 228).

Compared to incremental development and iterative enhancement, both discussed above, the difference in the evolutionary delivery is its explicit emphasis on delivering each increment to a customer (Gilb 1988, pp. 99, 274), and on using an existing system as a basis for the development of a new system (Gilb 1988, p. 229). Typical of this particular model is also focus on very small increments, and the long-term performance and adaptability of the resulting software product. Proceeding with very small increments which are delivered to a user for experimentation is made possible by the use of an existing system as a starting point (Gilb 1988, pp. 86, 100) and guarantees that possible mistakes and misunderstandings can be corrected while they still are small and easier to handle (Gilb 1988, p. 92). Using an existing system as a base reduces risks as it is riskier to build a totally new software system than to modify a small part of an existing one, when also user testing is easier to carry out (Gilb 1988, p. 84). As to the advantages and disadvantages of this model, these can be derived from the ones of incremental development. However, the characteristic features of evolutionary delivery must be taken into consideration.

3.3.3 Spiral model

Spiral model was first proposed by Boehm (1988) and has been further developed in, for example, (Iivari and Koskela 1987), (Iivari 1990), (Boehm, Egyed et al. 1998), and (Boehm and Belz 1990). This model attempts to incorporate the strengths of other models (for example, prototyping and waterfall model) and to resolve many of their difficulties (Boehm 1988). The model presents the software process as a spiral, where each of the loops can be considered to represent, for example, one fundamental step of the software process (Sommerville 2001, p. 53). Thus, the innermost loop might be concerned with requirements engineering, the next with design and so on. The major difference between spiral and other models is based on the spiral model's explicit focus on risk minimization, which is a very important project management activity (Sommerville 2001, p. 54). The spiral model assumes a risk-driven approach to the software development rather than a primarily document-driven (waterfall) or code-driven (prototyping) approach (Boehm 1988).

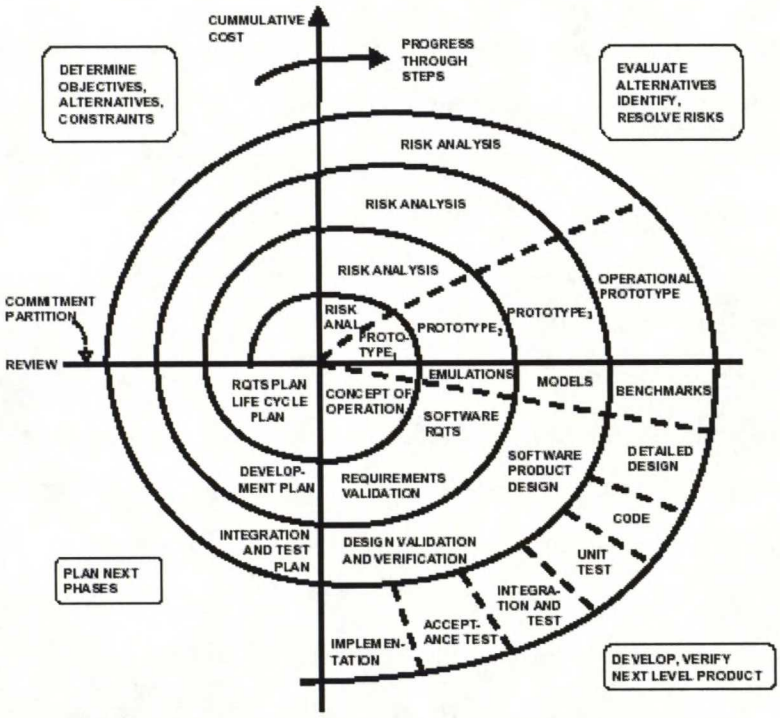


Figure 13. Software process using spiral model (Boehm 2000). The model has been further developed in (Iivari and Koskela 1987), (Iivari 1990), (Boehm, Egyed et al. 1998), and (Boehm and Belz 1990).

In spiral model each cycle results in a new intermediate product and the techniques used in developing the product of a particular cycle are a function of the risks present at the time of development (Davis and Sitaram 1994). Each cycle incrementally increases the systems degree of definition and at the same time decreases its degree of risk (Boehm 2000). In

general, each cycle involves four main activities (Boehm, Egyed et al. 1998), which are, in principle, iterated in each cycle. These activities are discussed below.

Each cycle begins with identification of the objectives to the software process and the product to be developed, elaboration of the alternative ways of achieving these objectives, and definition of the constraints imposed on each of these alternatives (cost, schedule, et cetera) (Boehm 1988; Boehm, Egyed et al. 1998). The second activity involves evaluation of the identified alternatives with respect to constraints and objectives (Boehm 1988). This should lead to identification of major sources of product and process risks (Boehm 1988; Boehm, Egyed et al. 1998). Each of the identified risks is analyzed in detail and a cost-effective strategy is formulated in order to resolve the sources of risk (Boehm 1988). Risk reduction measures to be taken may involve, for example, prototyping, simulation, benchmarking, analytic modeling, or combinations of these (Boehm 1988).

During the third activity, the process model for the software product to be developed is chosen (Sommerville 2001, p. 53) and the development of the product of the current cycle is carried out (Sommerville 2001, p. 55). After the deliverables for the cycle have been developed they are verified for correctness (McConnell 1996, p. 141). The final and fourth activity completes the cycle by reviewing the project and all the products developed during this particular cycle (Boehm 1988). This analysis leads to a decision on whether to continue with a further cycle or to terminate the project (Sommerville 2001 p. 54). If a decision is made to continue with the project, plans are drawn up for the next cycle (Sommerville 2001 p. 53). The objective is to ensure that all concerned parties are mutually committed to the approach for the cycle as planned (Boehm 1988).

The selection of an appropriate process model during the third activity is based on the relative magnitude of the risks and the relative effectiveness of the different process models in resolving them (Boehm 1988). If, for example, user interface risks dominate, the model chosen might be evolutionary prototyping and consequently, a series of prototypes might be developed in the subsequent spiral cycles (toward the right in the figure) (Boehm 1988). On the other hand, if the risks of budget and schedule predictability dominate, the waterfall model might be chosen in order to proceed with successive requirements cycle, design cycle and so forth (Boehm 1988). Hence, risk consideration allows the spiral model to accommodate any appropriate mixture of specification-oriented, prototype-oriented, et cetera process models

and may lead to a particular project implementing only a subset of all the potential spiral model steps (Boehm 1988). As a result of planning and analysis, different projects following the spiral model may implement different software processes (Boehm 2000).

A refined version of the spiral model (called win-win spiral model) adds three activities in front of each cycle 1) identify system's key stakeholders 2) identify their win conditions for a system 3) negotiate win-win reconciliation of these win-conditions (Boehm, Egyed et al. 1998). This refinement explicitly takes the software system's key stakeholders into consideration and encourages the participants to negotiate mutually satisfactory specifications.

The spiral model provides the developers with early indications of risks that can not be resolved (McConnell 1996, p. 143). It also focuses on eliminating errors and unattractive alternatives early by incorporating such issues as risk analysis, validation, and commitment (Boehm 1988). Moreover, the risk driven approach of spiral model helps the developers to decide the acceptable level of effort when, for example, analyzing requirements (Boehm 1988). The level of effort is simply determined by the level of risk incurred by not doing enough (Boehm 1988). Apart from the risk management perspective the advantages of the spiral model are that it allows uncertainty and midcourse changes, and provides progress visibility for management and customer (McConnell 1996, p. 156).

The disadvantages of this model include the fact that it is complicated to apply and thus, involves increased project planning and tracking effort, and requires attentive management (McConnell 1996, p. 143). The model may also involve problems with contracting in situations, where the same level of flexibility than in internal software development is not available and cycle-by-cycle commitments are not so easy (Boehm 1988). Problematic is also the model's assumption of the availability of risk assessment expertise (Boehm 1988).

As was discussed above, the spiral model encompasses other process models (Sommerville 2001, p. 55). In appropriate situations the spiral model actually becomes equivalent to, for example, the waterfall model (Boehm 1987), as each cycle of the spiral can become mapped to one phase of the waterfall model. In some other situations spiral model provides guidance on the best mix of existing process models (Boehm 1987). For example, one cycle may involve prototyping in order to resolve risks and this may be followed by the waterfall model (Sommerville 2001, p. 55). All in all, the spiral model is particularly applicable when very

large software systems are being developed (Boehm 1988). Even partial implementations of the spiral model are very helpful in overcoming major sources of project risk (Boehm 1988). Thus, parts of the spiral model can be used in conjunction with other process models as a risk reduction step before proceeding, for example, with the traditional waterfall model.

4 APPLICATION AND SUMMARY OF HIGH-LEVEL PROCESS MODELS

4.1 Identifying an appropriate high-level process model

Different high-level process models describe different approaches to and aspects of software development. On the other hand, different software development organizations, software products to be developed, and business environments set different requirements for the software process. Consequently, no single high-level process model can meet the needs of all software development projects, even within a single organization (McCormick 2001). Existing processes must be continuously changed to match the changing circumstances (Ward, Fayed et al. 2001) and a close attention must be paid to unique needs and goals of the software development organization and the type of the software product being developed when choosing a high-level process model for a given project if the software development effort is to be used efficiently (McConnell 1996, p. 465). For example, Alexander and Davis (1991) note that the selection of an inappropriate process model may result in a product that does not satisfy user needs (using waterfall when requirements are not understood), or in increased cost and longer development times (using prototyping when requirements are already understood). They also suggest taking into consideration of various factors regarding the nature of development personnel, project, resources, organization, and problem, when selecting a process model to be applied (Alexander and Davis 1991).

Interestingly, often the optimal high-level process model is not a pure waterfall, incremental, et cetera model, but a hybrid of various process models. It has been even argued, that in reality elements of each high-level process model are likely to be found in every software development project (Fairley 1985, p. 53). This is especially true for the development of most large software products where there is a need to use different approaches to different parts of the software system (Sommerville 2001, p. 51). Hence, instead of a single software development approach, a rich toolkit of process patterns along with guidelines for how to combine them to customize an approach for any given project is needed (McCormick 2001). Through deliberate customization the projects can be provided with the flexibility they need to succeed while at the same time balancing the needs of the projects against the strategic goals of the enterprise (McCormick 2001).

The need for hybrid models has been recognized by, for example, Boehm and Belz (1990) who further developed the spiral model in order to use it as a process model generator. As the

steps of the spiral model were originally applied to the software product, they now are applied to a software process to achieve a process model tailored to a particular project's characteristics. The process model generation begins with the determination of the project objectives and constraints, that is, for example, schedule, budget, end-product robustness, available technology, architecture and requirements understanding et cetera, and then continues with the identification and evaluation of process model alternatives with respect to these objectives and constraints. After this, the possible risks are analyzed and the resulting risk analysis is used to determine the project-tailored process model, that is, the best mix of process model alternatives is integrated to satisfy the project's needs. This model generation approach can be applied to a whole software process as well as be restricted to some specific part of it. (Boehm 1989; Boehm and Belz 1990)

Eventually, it is important to remember that identifying an appropriate process model whether pure or hybrid is only the beginning of the high-level process model application. After finding an appropriate process model, the model must be customized to the circumstances where it is applied by identifying the relevant activities and relationships among them. Also, if the process proposed by the model is to be used again and again, the basic rules for tailoring the process for different situations must be prepared and documented. Moreover, the guidelines on how to improve the process must be defined and an acceptance for the process obtained. (Whitten 1995, pp. 18-33)

Additional issues to be taken into consideration when the high-level process models are applied in practice are concurrency and reuse in software process. These approaches can be applied to support all high-level process models introduced above and are discussed next.

4.2 Concurrency in software process

Concurrency approaches can be used to support the high-level process models in project status tracking. It has been argued that concurrency of activities is inherent in all non-trivial software development and that tracking software development project status in terms of the major phases of, for example, the pure waterfall model gives no idea of the true status of the projects (Davis and Sitaram 1994). Concurrent development proposes that the different phases or entities of the software process can exist simultaneously, and that different states of these

phases or entities should be used to track project status. This kind of approach has been proposed in, for example, Humphrey and Kellner (1989), and Davis and Sitaram (1994).

The entity process approach presented by Humphrey and Kellner (1989) proposes, that in a software process there are entities that are persistent throughout a software development project. Each entity has a defined sequence of states, that is, some externally observable modes of behavior (Pressman 2000, p. 39), through which it passes during its lifetime (Humphrey and Kellner 1989) and also specific triggers that cause the transitions between these states (Humphrey and Kellner 1989). Thus, by undergoing many transformations the entities survive throughout the process (Humphrey and Kellner 1989). Consequently, entities like requirements specification or software design should not be seen as end results of respective phases that will not undergo further changes, but as entities that coexist and evolve throughout the whole development project by changing the states they are in. This helps keeping the entities up-to-date throughout a software process (Humphrey and Kellner 1989).

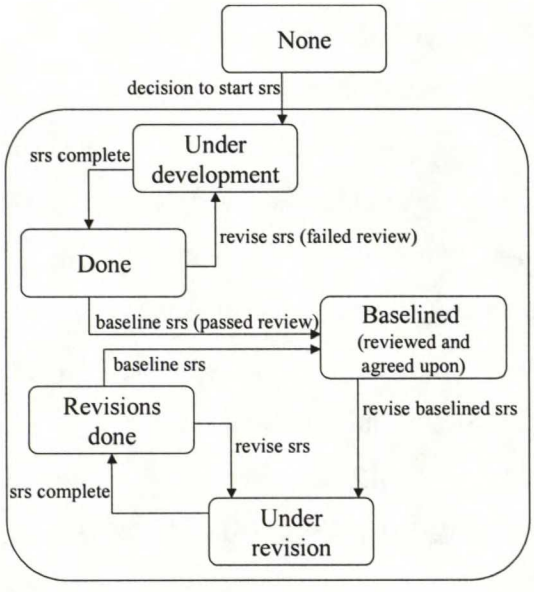


Figure 14. Alternative states of software requirements specification (srs) activity (Davis and Sitaram 1994).

As entity process approach proposes concurrency of entities, concurrent process approach proposed by Davis and Sitaram (1994) focuses on concurrency of software development activities. Davis and Sitaram introduce that there is a sequential initiation of each activity, but once initiated the activities are performed concurrently. Activity is here defined as a particular software engineering activity typically associated with some particular phase. Examples of activities might be requirements specification, preliminary design, or system testing. Like the

coexistence of entities in the entity process approach, the coexistence of activities associated with different phases is achieved through different states the activities are in (Davis and Sitaram 1994). For example, requirements specification may be in state base-lined, while at the same time activities preliminary design and system testing may be in states under revision and awaiting to be revised respectively.

The concurrency in the above approaches can be represented with a series of activities or entities and their associated states. Through these state–based representations, that embody the true process behavior and can be refined to progressively greater levels of detail, a better understanding and more accurate control of project status are possible (Humphrey and Kellner 1989). Further, Blackburn and Hoedemaker (1996) argue, that concurrency in software process can produce great enhancements in regard of speed, quality and cost.

4.3 Reuse in software process

The fact that complex, high-quality software products must be built in very short time periods together with the demand for lower software production and maintenance costs has given raise to a systematic component reuse in software development (Pressman 2000, p. 738; Sommerville 2001, pp. 49, 307). This approach, sometimes called component-based software development (CBSD), tries to avoid building software system from scratch and relies on a large base of reusable software components and some integrating framework for composing the components into software systems (Sommerville 2001, pp. 44, 49). Examples of the components reused are domain knowledge, specifications, designs, program code, and test data developed for the past projects and similar to the components needed in the current project (Isoda 1991; Pressman 2000, p. 123). The components reused may be commercial-off-the-shelf components (COTS) as well as internally developed (Pressman 2000, p. 739).

Systematic component reuse requires a software process that explicitly considers how existing components may be reused (Sommerville 2001, p. 307). Besides the environment encouraging component reuse there needs to be an up-to-date component library system in order to enable the easy retrieval of the existing components (Pressman 2000, p. 284). Further, to help the use and modification of the components, the components must be documented as to where and with what problems they have been used (Sommerville 2001, p. 308).

In the reuse-oriented software process the requirements specification phase is followed by the search of potentially applicable components to implement the specification (Redwine and Riddle 1988; Pressman 2000, p. 739; Sommerville 2001, p. 50). The potential components are qualified to ensure that they properly perform the function required, fit into the architectural style specified for the system, and exhibit the quality, performance, reliability and usability characteristics required (Pressman 2000, p. 739). As there often is not an exact match between current project's requirements and the available components, an attempt to modify or remove requirements that cannot be implemented with existing components usually follows (Pressman 2000, p. 740; Sommerville 2001, p. 50). If requirements cannot be modified new components must be developed (Pressman 2000, p. 740; Sommerville 2001, p. 50) and existing components modified (Redwine and Riddle 1988). After this, a framework of the system is designed or an existing framework is reused (Sommerville 2001, p. 50) to bind the components together and thereby form an operational software system (Redwine and Riddle 1988; Pressman 2000, p. 739). Finally, the software system developed is validated and verified (Redwine and Riddle 1988; Sommerville 2001, p. 50).

As was already discussed above, the components are, classically, plugged into a skeletal software infrastructure that handles the communication and coordination among the various components (Clements 1995). These skeletal infrastructures, or product platforms, consist of a set of subsystems, and interfaces between these subsystems and the external environment (Meyer and Seliger 1998). Recently, also this coordination infrastructure itself has become considered as a reusable component that is potentially available in pre-packaged form (Clements 1995). The platform forms a common structure, a base software engine, from which a stream of derivative products can be efficiently developed and produced (Meyer and Seliger 1998). These derivative products are add-in modules that utilize the infrastructure and information provided by the platform and can be seamlessly plugged into it (Meyer and Seliger 1998). Now, by building small solution-specific add-in modules and plugging them into platform that provides the basic functionality for all specific solutions, products tailored for particular niches can be developed more rapidly and flexibly without starting from zero every time (Meyer and Seliger 1998). These new products can be introduced either as new versions of existing products or as entirely new products or systems (Meyer and Seliger 1998)

Advantages of reuse are manifold. Apart from reducing the costs and the amount of components to be specified, designed, implemented, and validated (Sommerville 2001, p.

307), reuse usually leads to faster delivery of the software product (Clements 1995). Reuse also increases predictability and thereby leads to a higher quality (Pressman 2000, p. 738) and reduced risks (Sommerville 2001, p. 308). Also the reliability is increased as the components are used and tested in a variety of working systems and environments (Clements 1995). Further, the product can be more easily adapted to evolving requirements (Nierstratz, Gibbs et al. 1992). Problems of the reuse, in turn, include requirements compromises which are inevitable if extensive reuse is wanted and may lead to a product that does not meet the real needs of the customer (Sommerville 2001, p. 50). Also, if COTS components are used, some control over the system evolution is lost as new versions of the system components are not under the control of the organization using them (Clements 1995). Reuse may also lead to increased maintenance costs in situations where the source code for components used is not available (Sommerville 2001, p. 309). Further, integrating software development tools with a component library may be difficult (Sommerville 2001, p. 309). Finally, maintenance of the component library can turn out expensive (Sommerville 2001, p. 309).

4.4 Summary of the high-level process models

This section presents a synthesis of the high-level process models based on the literature review made for this study. To begin with, table 2 attempts to summarize the high-level process models discussed in this study and gives indications of the nature of the support provided by these models in specific areas of the software process. The capability areas in the table 2 are selected so as to bring forth the differences between the discussed models and include the model's capability to:

- Support requirements elicitation and evolution, that is, does the process model take into consideration the difficulty of stating requirements explicitly in the beginning of the development project by supporting user experimentation and feedback? This affects the degree to which the evolving customer requirements can be taken into consideration during the development and incorporated into the resulting software product.
- Produce clear and modifiable software, that is, does the model strive for software modifiability and clarity through systematic consideration of design activities, controlled design modifications and systematic documentation? This affects the ease of incorporating the proposed changes and maintainability and solidness of the software product.
- Provide progress visibility, that is, does the model provide internal and external progress visibility in form of operational software early in the process, early and incremental (internal and external) releases, and steady phasing and steady documentation? This affects the degree

to which the customer and management can track the progress of the project and to which the customer can make use of the product under development.

- Manage risks, that is, how are risks taken into consideration by the model?

Capability to:	Model:	Code-and-fix	Waterfall (Royce)	Evolutionary prototyping	Throw-away prototyping	Incremental development	Iterative enhancement	Evolutionary delivery	Spiral model
Support requirements elicitation and evolution		Requirements specification not planned for, iteration incorporated (Boehm 1988, ; McConnell 1996, p. 140)	Requirements specification carried out early in the process and fixed thereafter (Royce 1987)	Requirements specification carried out iteratively and evolves gradually based on user experimentation and feedback (Sommerville 2001, p. 172)	During prototyping the requirements specification carried out iteratively and evolves gradually based on user experimentation and feedback (Sommerville 2001, p. 172), * thereafter maybe more or less fixed	Requirements specification carried out early in the process. Thereafter may to some extent gradually evolve based on user experimentation and feedback (Sommerville 2001, p. 52)	Requirements specification carried out iteratively and evolves gradually based on user experimentation and feedback (Basili and Turner 1975)	Requirements specification carried out iteratively and evolves gradually based on user experimentation and feedback (Gilb 1985)	* Model evolves to support evolving requirements as needed (Boehm 1988)
Produce clear and modifiable software		Design not emphasized, uncontrolled design modifications, documentation not emphasized (Boehm 1988)	Design emphasized, design modifications in principle not accepted but well-planned if done, documentation emphasized (Royce 1987)	Design not emphasized, design modifications common but not comprehensively planned for, documentation not emphasized (Gordon and Bieman 1995) (Sommerville 2001, p. 177, 47)	Design not emphasized, design modifications common but not comprehensively planned for (Sommerville 2001, p. 47), * after prototyping depends on the model applied	Design emphasized, design modifications comprehensively controlled, documentation emphasized (Sommerville 2001, p. 178)	Design emphasized, design modifications comprehensively controlled, documentation emphasized (Basili and Turner 1975)	Design emphasized, design modifications comprehensively controlled, documentation emphasized (Gilb 1985)	Model evolves to support systematic design, controlled design modifications and documentation as needed (Boehm 1988)
Provide progress visibility		Early signs of progress in form of operational software, steady documentation or phasing not incorporated (McConnell 1996, p. 140)	No signs of progress in form of operational software until the end of development process, steady documentation and phasing incorporated (Royce 1987)	Early signs of progress in form of operational software, steady documentation or phasing not emphasized (McConnell 1996, p. 147; Sommerville 2001, p. 46-47, 172)	Early signs of progress in form of operational software (Sommerville 2001, p. 172), * possible early delivery of useful product subsets * steady phasing and documentation more or less incorporated	Early and steady signs of progress in form of operational software, early and incremental delivery of useful product subsets, steady phasing and documentation incorporated (Sommerville 2001, p. 178)	Early and steady signs of progress in form of operational software, early and incremental delivery of useful product subsets, steady phasing and documentation incorporated (Basili and Turner 1975)	Early and steady signs of progress in form of operational software, early and incremental delivery of useful product subsets, steady phasing and documentation incorporated (Gilb 1985)	* Steady review points, model evolves to support steady phasing and documentation, to provide early signs of progress in form of software, and to support early delivery of useful product subsets as needed (Boehm 1988)
Manage risks		Risks not explicitly considered (McConnell 1996, p. 140)	Risks considered with prototyping step, comprehensive planning and limited iteration (Royce 1987)	Risk management focus on developing high-priority areas first and requirements elicitation (McConnell 1996, p. 434; Sommerville 2001, p. 172)	* This model is essentially a tool for risk assesment (Sommerville 2001, p. 172, 179)	Risk management focus on developing high-priority areas first (Sommerville 2001, p. 52)	Risk management focus on developing high-priority areas first (Basili and Turner 1975)	Risk management focus on developing high-priority areas first (Gilb 1985)	Comprehensive risk management incorporated (Boehm 1988)

Table 2. Capability summary of the presented high-level process models.

The asterisk (*) that can be found in the column "throw-away prototyping" means that the degree to which the capability concerned is provided by this model depends on the process model applied in the development of the final production quality product. As discussed earlier, throw-away prototyping itself is only a springboard for the other process models. In the column "spiral model", on the other hand, the asterisk reminds the reader of the spiral model's inherent ability to respond to the requirements set by the environment by becoming equivalent to other, appropriate process models. Thus, in principle, the spiral model possesses all the capabilities to the highest possible degree by always becoming equivalent to the best possible model.

From the table it can be seen that different high-level process models differ in their capabilities to provide support in the abovementioned areas of requirements elicitation and evolution, software modifiability and clarity, progress visibility, and risk management. Different software processes, in turn, differ in their needs for these areas (see figure 15). Consequently, as the appropriateness of a given high-level process model for a given software process is being assessed, the capabilities of the process model and the needs of the software process in question should be taken into consideration. If there are deficiencies in the model's capability to support a certain important area, another model can be chosen or a proper supporting mechanism applied in order to adequately provide the capability in question.

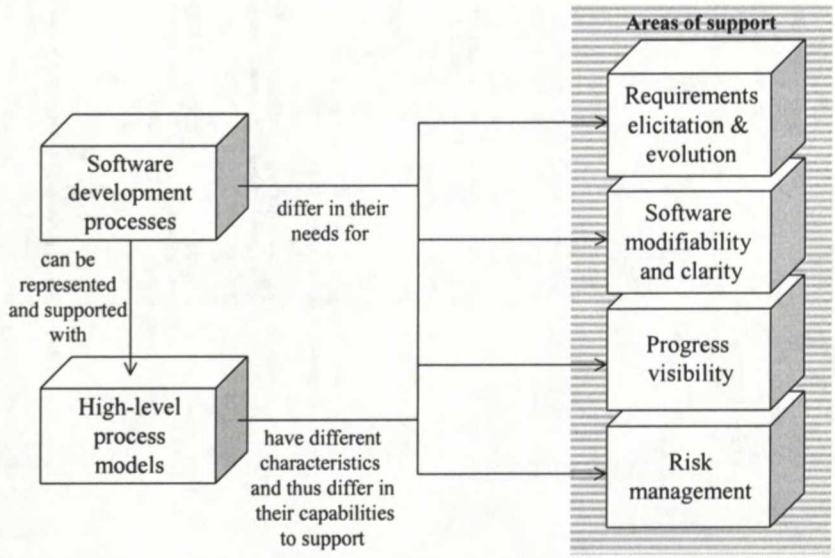


Figure 15. Different software processes differ in their needs and thus, need to be supported with and implement different high-level process models.

Apart from using the abovementioned support areas as a tool in process model selection, these areas can be used as a tool for software process analysis when reasons for the appearance of a given software process are being explored. That is, the role of the support areas in a given software process reflects the needs of that process and affects the way the process looks like in terms of high-level process models. However, the fact that a given software process resembles some specific high-level process model can be either a result of deliberate up-front decision to use a specific model or just a result of intentional gradual adaptation to meet the needs as they emerge.

The figure 16, in turn, attempts to position the discussed high-level process models and acts as a framework based on which the software processes can be analyzed in terms of high-level process models. The horizontal axis denotes the iterativeness versus linearity of the software process implied by the model, and thereby the model's attitude towards the software process as a learning process or manufacturing process respectively. Further, the nature of the iteration is illustrated, that is, iteration can be of more or less controlled nature. The process models with controlled iteration are stricter, stiffer, and more planning-oriented in their iteration than models with uncontrolled, even chaotic iteration. Like linearity, controlled iteration is likely to produce more modifiable and clear software than uncontrolled iteration. On the other hand, these models may have problems in quick adaptation to continuously evolving customer requirements. Uncontrolled iteration, in turn, is probable to lead to problems regarding the modifiability and clarity of resulting software and thus, difficulties in incorporation of the requested changes during the software development and later during the maintenance. In-between models, then, balance the needs for requirements evolution and modifiability by facilitating quick iteration with some degree of planning.

The vertical axis, then, presents the number of deliveries suggested by the model. Models with multiple deliveries during the process imply early operational software with a possibility for incremental delivery to a customer and thus, provide good progress visibility in form of operational and useful software subsets for both management and customer. Further, the models with multiple deliveries have been divided into those that focus on delivering each increment to the customer (outwards) and those (in/outwards) that see this as a possibility but may as well use increments as an internal development approach only. As for the models with one delivery at the end of the project, these can provide progress visibility in form of operational software either early in the project or late in the project. From the customer's point

of view early operational software without delivery can, for example, mean experimentation with the existing implementation. Further, in models that provide early operationality with one delivery, the "entire solution" is initially implemented and then iteratively refined, whereas models with multiple deliveries imply division of the solution into pieces and iteration based on the experimentation with these pieces.

As for the spiral model and throw-away prototyping, these models can accommodate the other process models, and are marked with a little star at the corner of the other process models, except for the code-and-fix model. The application of these models can be considered to demonstrate an organized and systematic approach to software development and thus, should not lead to the application of the non-systematic code-and-fix model.

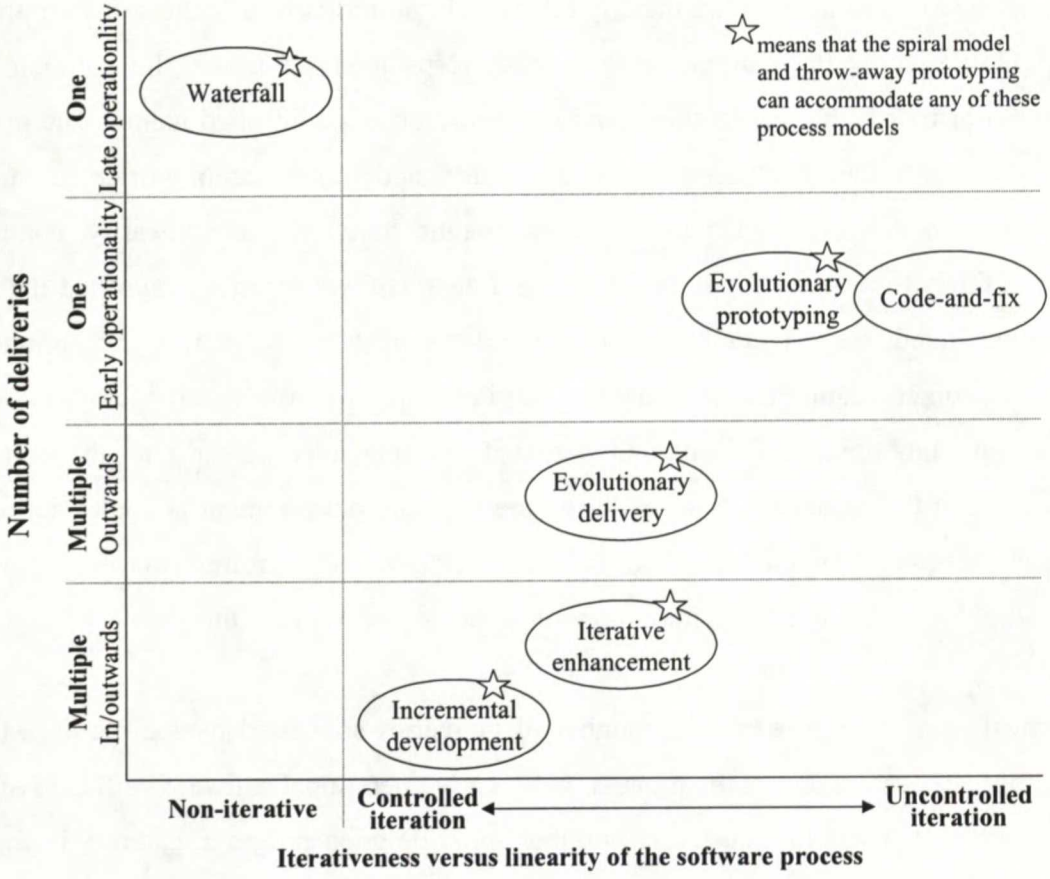


Figure 16. Framework for positioning high-level process models.

To sum up, the literature review on high-level process models revealed that the discussion on high-level process models began with the recognition of the deficiencies of an unorganized code-and-fix software process. While the code-and-fix model was sufficient for the development of small software products, it did not scale up well. Hence, as software products

grew in size, a more organized and planning-oriented approach to software development was needed. A systematic, non-iterative process model, that is, waterfall, was proposed as a solution and thus, a move from one extreme to another was made. However, as the applications and thereby software then continued to grow in sophistication and at the same time the volatility of business environment continued to increase, the problems of the rigid waterfall model became apparent. Consequently, the idea of iteration with control and user experimentation began to evolve, and as a result, incremental and prototyping models emerged. The differences between the models incorporating both iteration and control arise from their differing emphases on planning versus experimentation. As, for example, incremental development emphasizes planning orientation with some level of experimentation the focus of evolutionary prototyping is more on experimentation with some level of planning. However, even evolutionary prototyping incorporates planning orientation to a degree that distinguishes it from the early chaotic code-and-fix model. The approaches like evolutionary delivery and iterative enhancement, then, consider experimentation and planning equally important.

Interestingly, the software development approaches are increasingly transitioning towards the somewhat planned iteration with multiple and ever earlier releases, as the importance of delivering useful, high-quality product subsets to the customer early and often in the software process has become more and more important. This development is reflected by, for example, evolutionary delivery discussed earlier but also as a growing interest towards so-called agile methodologies (see, for example, Fowler 2001; Abrahamsson, Salo et al. 2002). The importance of short iterations and early continuous deliveries has but grown and overcome the idea of preemptive documentation, shifting the focus from conforming to original plan to satisfying the customer at the time of delivery (Highsmith and Cockburn 2001; Boehm 2002). Early operationality combined with short iterations and continuous deliveries provides managers and customers with concrete signs of progress and moreover, provides quick feedback and a possibility to continuously adjust to new information. Although changes throughout the development project lifecycle (even at the very end of it) are accepted as inevitable, importance of planning is not forgotten - it is just that the focus is more on the process of planning instead of the resulting documentation (Highsmith and Cockburn 2001). Quality of the software product is also stressed although the change is embraced and design is done in small chunks (Highsmith and Cockburn 2001).

All in all, both experimentation-oriented and orderly planning-oriented (or even linear) approaches have a homeground in which each clearly works best and where the other will have difficulties. Experimentation-oriented approaches excel in turbulent environments where requirements are not pre-specifiable, and where comprehensive pre-specifications nothing but overconstrain the development team and provide a source of major rework and delay. Orderly planning-oriented approaches, in turn, work best when the requirements and other important factors are relatively stable and can be determined in advance. Moreover, thorough planning is vital for stable, safety critical software systems. In-between approaches, then, are needed for projects that combine a mix of experimentation- and planning-oriented homeground characteristics. (Boehm 2002)

To conclude, in addition to revealing the confusion of concepts and terminology prevailing in the software development discourse in general, the literature review made on the high-level process models showed that there is no consensus on the categorization and naming of high-level process models, not to mention the exact origins of the various models. Unambiguous and uniform ways of categorizing and naming different high-level process models simply seemed not to be striven for, instead, the focus was on proposing different approaches in order to tackle the problems prevailing in software development at the time. The process models seemed to stem from the hands-on-experiences of the authors and thus, process models encompassing similar features and with some or virtually no differences have sometimes been proposed by many different authors relatively simultaneously. One reason for this kind of confusion may be the inherent complexity of the software development field and the coexistence of several objectives and attitudes of mind as proposed by Lonchamp (1993). Interestingly, a comprehensive review of the various high-level process models seemed to be lacking. This study attempts to begin filling this gap in research.

5 GAME DEVELOPMENT

The empirical part of this study focuses on a development process of one specific type of software, that is, games. The game development was chosen as a topic because of the following reasons. First, game development projects have evolved from small-team efforts to industrial size projects and along this the complexity of game development processes is growing dramatically (Walton 1998; vom Scheidt 2000). Partly resulting from this development, the games industry has been experiencing the typical process related problems of software development (Walton 1998; vom Scheidt 2000) and thus, importance of software process definition and modeling is being increasingly acknowledged also in game development. Second, unlike the research on the psychological, health and other impacts of games, the research on game development is, for the time being, still only taking shape. This study aims to contribute to this early stage of the game development research by exploring the game development as a software process in a Finnish mobile game development company.

Chapter five serves as an introduction to the game development and thus helps comprehend the context of the empirical part of this study. The section 5.1 briefly introduces the growing societal impact of the games industry, and in section 5.2 a typical game genre categorization is discussed. Section 5.3, on the other hand, discusses the generic game development process, and introduces the typical roles and phases in game development project. Further, the problems brought by and solutions proposed to the growing process complexity in game development are discussed.

5.1 Introduction to games industry

People today seek experiences and entertainment in which they are fully involved and in control. This type of entertainment has been delivered to consumers through game software (hereinafter games) for more than 25 years, and as a result of continuous advances in technology, the possibilities for improving game design, graphics, and the complexity of stories and thereby providing people with ever more enchanting experiences are continuously growing. Consequently, the games appeal to an ever-expanding consumer base and growing impact of games industry on society seems more than likely. What is more, as games continue to penetrate popular culture, they more and more overlap with other entertainment genres, such as movies, television, and music. (IDSA 2001b)

The games industry can be seen to consist of two major sectors: the video games market and the computer games market. The video games market began to take shape when electronic games entered the consumer market with an introduction of Atari's Pong arcade game in the early 1970s. Only a year after this, the first home video game system was launched by Magnavox and followed by many other competing systems. The modern-day video game industry, however, is often considered not to have taken shape until 1985, when Nintendo introduced its Nintendo Entertainment System (Famicom). Since then, new game consoles with enhanced features have been progressively introduced. The computer games market, on the other hand, became established by 1984 as a result of the release of affordable and programmable home computers. By the end of the decade, these were replaced by the growing use of IBM-compatible home PCs and the cheaper and more games-oriented computers provided by, for example, Commodore and Atari. Eventually, the market became dominated by PC, as the other computer games platforms failed to evolve at the same rate with it. (Games Investor 2002)

Until the early 90's, development in both markets was largely determined by the shelf lives of the individual games hardware standards of the time. After that, the overall games market has begun to stabilize. The computer games market has showed a strong growth and, as was already discussed above, become dominated by a single, open and ever evolving, hardware standard, the PC. In contrast, the video games market has been dominated by proprietary standards, currently Sony's Playstation 2, Nintendo's Game Cube and Microsoft's Xbox. Due to the closed nature of the video game standards and the reluctance of the standard owners to release major hardware upgrades to prevent obsolescence, the shelf lives in video game market are expected to stay limited for the foreseeable future. (Games Investor 2002)

Interestingly, an ever more important category of games is online games, that is, games that are played with a device connected to the Internet so that the player can play against other players similarly connected to Internet with an online-enabled game device. Online games broaden the potential consumer base by putting games on new platforms like mobile phones and interactive TV. As for the term mobile games, this is generally used to refer to computer games accessed through mobile terminals such as handheld games consoles, personal digital assistants (PDAs), and mobile phones.

Mobile gaming started in the form of specialized handheld devices like the Nintendo Gameboy and its predecessors, and has more recently been bundled with mobile phones. The first mobile phone games were embedded single player games such as snake. The development continued with SMS (where individual short messages were sent) and WAP (text-based markup language similar to HTML with limited graphics) based mobile phone gaming. Ever since, the development has continued with generations of ever more sophisticated, faster and powerful internet capable mobile phones bringing the opportunity to provide player with ever more enchanting experiences. Once the "permanently online", positioning-enabled phones really have taken their place, the wireless gaming is expected to be taken for a real spin.

The growing importance of games industry has already been clearly noticed. For example, the research commissioned by the Interactive Digital Software Association (IDSA) showed major impact of games industry on U.S. Economy. A detailed study of economic data showed an annual growth of 14.9 % in sales in the game software industry between 1997-2000. This was more than double the annual growth rate of the U.S. economy as a whole (6%) and far outpacing the annual growth in sales in the related industries over the same period. As for the growth in wages and employment, even motion picture production, distribution and allied services sectors, considered to be America's leading entertainment industries, have recognized a lower annual growth during the same period. (IDSA 2001a)

In year 2001, U.S. sales of the computer and video game software grew 7,9% to $6.35 billion (IDSA 2002). The amount of video and computer games sold rose 4.5% to 225.1 million units compared to year 2000 (IDSA 2002). Moreover, for the first half of 2002, the dollar sales of both video and computer games in U.S showed a significant revenue growth (NPD Techworld 2002). As for the future, the global computer and video games industry is expected to experience a strong 15-25% compound annual growth between 2002-2005, driven principally by the release of new console hardware (Sony's PlayStation 2, Microsoft's Xbox, Nintendo's Game Cube, and Game Boy Advance) (Games Investor 2002). As for the mobile gaming over cellular network, this is predicted to be worth U.S$ 4,3 billion worldwide by 2006 (Ovum 2001).

5.2 Game software

Different games vary greatly in terms of, for example, complexity and visuality, and hence, in size and other characteristics. As there, consequently, is no such a thing as typical game software, characterizing games in general in terms of, for example, software features presented in section 2.2, or comparing game software to other software application types is not meaningful. However, inside the game software category, it is common to differentiate between the different game types or genres. The following genre categorization is commonly used (Bates 2001, pp. 7-14):

- Adventure games are story-based games, often relying on puzzle solving to move the action along. There is a large complex world for the player to explore, along with interesting characters and a good story. One example of this is King's Quest.
- Action games are essentially real time games where the player intensively interacts with what is happening on the computer screen. The genre is mainly dominated by first person shooters (FPS) such as Tomb Raider in which the player sees the hero or heroine moving through the environment.
- Role-playing games (RPGs) are typically based on the player directing a group of heroes on a series of quests. The game is about gradually increasing the abilities and strengths of the heroes. An example of a classic RPGs is Ultima.
- Strategy games require players to manage a limited set of resources to achieve a predetermined goal. An example of this is Command and Conquer.
- Simulations seek to emulate the real-world operating conditions of complicated machinery, such as tanks. The equipment controls in the game can either be simplified or extremely sophisticated, the focus being on either get-in-and-play or hours-of-learning respectively.
- Sports games give the player a possibility to participate in their favorite sport as a couch or a player.
- Fighting games are two person games in which player(s) controls figures on the screen and uses a combination of moves to attack the opponent and to defend against the opponent's attack.
- Casual games are adaptations of traditional games such as chess and bridge, or television quizzes and trivia.
- God games have no real goal except for encouraging the player to fool around with the game only to see what happens. One example of these is The Sims.
- Educational games teach and entertain the player at the same time.
- Puzzle games provide intellectual challenge of problem solving. Puzzles are an end in themselves, and not integrated into a story as is typical in adventure games.

5.3 Game development

5.3.1 Different roles in a game development process

The roles typically related to the different aspects of game development include the following:

Vision: It is essential in a game development project that there is a person who has the complete vision of the game and knows throughout the chaos of development how all the pieces eventually come together to serve this vision. This person acts as a gatekeeper through whom all new ideas must pass. (Bates 2001, p. 145)

Production: Producers in general manage risks and take care that the game aligns with the company's goals. In a game development effort, there may be two producers: one representing the game development company (internal producer) and other representing the publisher (external producer). External producer oversees the game development to ensure that the game gets delivered on time, on budget, and with agreed features. The external producer tracks project milestones, approves payments to developer, and intermediates the publishers wishes to the developer and vice versa. Internal producer (often called project manager or project lead), in turn, manages the game development team directly and represents it to the outside world such as publisher, sales and marketing department. Internal producer, for example, makes the decisions on the resources for the development effort, and manages change. (Bates 2001, pp. 146-151)

Design: In game design, the typical roles include game designer, level designer, and writer. The game designer is responsible of the overall feel of the game and thus, is likely to be the vision guy through whom all new ideas must pass. The game designer creates the game design document and keeps it up-to-date throughout the development. Apart from that, game designer often doubles as a writer, directs the level designers, and advices PR, marketing and sales in their promotion efforts. Level designers, in turn, work with artists and programmers to outline the smaller sections of the game. Writer designers write game dialogue and design other parts of the game where words are needed. (Crosby 2000; Bates 2001, pp. 90, 154, 159).

Programming: The main roles in the realm of game programming include tech lead and programmers. Tech lead is responsible for creating the game architecture, instructing the

development team of what is technically achievable, evaluating and acquiring the technology necessary for the game development, and understanding and explaining others the technical trade-offs. Tech lead also leads and instructs the programmers, sets coding standards and best practices, establishes version control procedures et cetera. Programmers, on the other hand, plan and write the game software and thus are the ones turning the game vision into executable code. (Bates 2001, p. 161-164)

Art: Art lead is responsible for the look of the game, and directs the people responsible for the creation of the art representing the game vision. The artists working in game development can be divided into concept artists, character modelers, animators, background modelers, and texture artists. Concept artists work with the designers to create the look of the game, and make sketches of characters and settings trying to bring the vision into life. Character modelers use modeling software to create 3-D models of the characters and other elements, to which animators then give life by making them move. Animators also create short movies that are played at predetermined times in the game. Background modelers, in turn, are responsible for the environments and settings where the game takes place. Finally, texture artists create skins for, for example, background and creatures. (Crosby 2000; Bates 2001, pp. 168-174)

Testing: In addition to ensuring that the game behaves as intended, testers also check that the game is fun to play. Test lead leads the testers, gives feedback on game features, and creates test plans, whereas testers do the actual the testing. (Bates 2001, pp. 174, 177)

Other, possibly external, resources in game development projects are used to provide the game with voice, music, sound effects, and user manual (Bates 2001, pp. 184-200). The roles presented above are roles directly involved with the game development per se. The organizational roles such as the ones involved with sales and marketing, or budgeting are excluded from the above list.

5.3.2 Generic game development process

Game software is produced through a software process by which the game vision is translated into a game software product. The following game development process phases have become standards across the games industry (Bates 2001, p. 206).

Concept development: During concept development phase, the main objective is to come up with an idea, decide what the game is about (high concept) and to write this down (Bates 2001, p. 206). The key elements to be manipulated during concept development are gameplay, scope, and technical risk (Bates 2001, p. 17). In other words, the decisions have to be made regarding the rigor and nature of the gameplay, size of the project in terms of time and money, and the use of new technology (Bates 2001, pp. 17-18). The documents created during this phase are game concept and game proposal (Ryan 1999a).

Game concept document expresses the core idea of the game and is intended to encourage the flow of the ideas. In this document, for example, the content and entertainment value of the game is described, the key features setting the game apart from others summarized, and genre and platform introduced. The game concept document helps decide whether the idea is worth proceeding with. (Ryan 1999a)

The main result of the concept development phase, however, is the creation of game proposal document (Bates 2001, p. 206). Game proposal is a formal project proposal used to secure funding and other resources for the game development project (Ryan 1999a). This document is an expansion upon and includes a revised version of the game concept (Ryan 1999a). Game proposal can be considered as an executive summary of the game to be accomplished and summarizing what the game is about and why it will be successful (Bates 2001, pp. 18, 206). In case of external funding, the game proposal's main goal is to get the go-ahead to proceed into and funding for pre-production phase (Bates 2001, p. 18). The approved game proposal acts as a basis for a project plan (Bates 2001, p. 18). Game proposal contains following sections:

- High concept: Is a short revised description of what the game is about and why it is different. High concept is valuable during the development phase because it suggests what features contribute to the game's main focus and thus helps to decide which features to include and which to leave out. (Bates 2001, p. 207)
- Features summary: Lists the major selling points of the game and features that will make the game exceptional (Bates 2001, p. 207).
- Story: Introduces the plot and the main character (Bates 2001, p. 207).
- Gameplay mechanics: States the game genre and briefly describes what the player does while playing the game (Bates 2001, p. 207).
- Art: Expresses the feel and purpose of the game, includes character sketches et cetera, and aims to get the publisher excited (Ryan 1999a).

- Target market description and competitive analysis (Bates 2001, p. 19).
- Target hardware platform (Bates 2001, p. 19).
- Technical analysis: Summarizes the experimental features, major development tasks, technical risks, and estimated resources (Ryan 1999a).
- Estimated schedule (Bates 2001, p. 19).
- Cost and revenue projections (Bates 2001, p. 207).
- Legal analysis: Copyrights et cetera contracts that possibly incur fees (Ryan 1999a).
- Risk analysis (Bates 2001, p. 209).
- The team: Previous experience et cetera. (Bates 2001, p. 20).

Pre-production (proof of concept): During pre-production phase the development team aims to demonstrate that the game is worth making and that the team is capable of making it and working as a good partner (Bates 2001, pp. 18, 209). If the publisher wants to proceed with the project the fees and royalties are negotiated, and development milestones and deadline set (Human Capital, report 2001). The products of pre-production phase are:

- Game design document: This document exhaustively details what goes into and will happen in the game (Bates 2001, p. 210). It can be considered as a functional specification fleshing out the skeleton of the vision expressed in the game concept and game proposal, and elaborating the vision in very clear terms so that the intent and function of different elements are conveyed (Ryan 1999a).The features included into this document become the requirements on which the technical plan is made (Bates 2001, p. 210). During the development phase the game design document always is the most current representation of everything there is to know about what the player experiences in the game (Bates 2001, p. 210). Moreover, development should not start until the requirements analysis is sufficiently detailed, that is, there is an unambiguous and thorough game design document (Crabtree 2000; Bates 2001, p. 221).
- Art bible: Describes the looks of the game and thus makes it possible to create consistent style for the game (Bates 2001, p. 210).
- Production path: Describes the methods and tools needed to gradually transform the game vision into reality (Bates 2001, p. 210).
- Technical design document (technical specification): This document describes how the game design is transformed into an actual software and what is involved in this transformation process (Bates 2001, p. 211). It describes how each area of the game's functional specification is to be implemented and how the game performs its intended function (Ryan 1999a).
- Project plan: Includes, for example, task lists and dependencies, and implies the schedule, resources, and milestones. Is revised and updated throughout the project. (Bates 2001, p. 212)

- Prototype: This is the tangible result of the pre-production phase, a working piece of software that captures on screen the essence of what makes the game exclusive (Bates 2001, p. 212). As a look-and-feel demonstration simulating what the real product will look like, it is very important for the publishers (Bates 2001, pp. 212-213). In addition to bringing into life the vision, prototype shows whether the production path is working (Bates 2001, p. 213).

Implementation (development): This phase typically lasts from six months to two years (Bates 2001, p. 213). If the implementation takes longer than two years, the game runs the risk of personnel turnover, and having features and technology become out-of-date, which in turn lead to rework and schedule delays (Bates 2001, p. 213). In case of an external publisher, the developers have contractual milestones meeting of which is precondition to getting paid (Bates 2001, p. 213).

Alpha testing, Beta testing and Code freeze: In the alpha testing phase the game is mostly playable from the beginning to the end and the focus starts to shift from building to finishing. In the beta testing phase, on the other hand, the development is stabilized, and the focus is primarily on bug fixing. The testing may include obtaining approval from the hardware manufacturer, and testing the game on different combinations of hardware. Finally, during the code freeze phase the preparation of candidate master discs begins and the only changes accepted are those that address the major bugs turning up in testing. (Bates 2001, pp. 215-218)

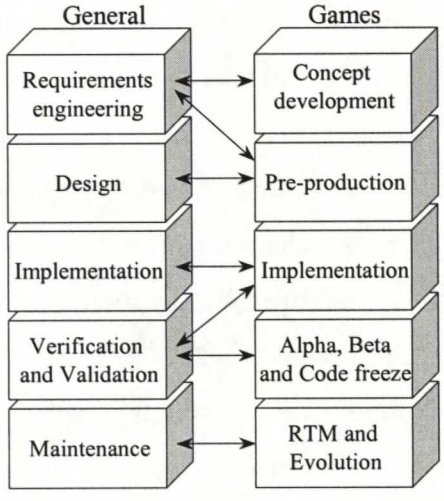


Figure 17. Simplified description of the correspondences between general software process and general game development process.

Release to manufacture (RTM) and Evolution: In RTM one of the candidate master discs has been accepted for manufacture and the game goes to production. As for the patches and upgrades, these are a form of post-release support. After release, the game may be patched, that is, fixes are released to remove problems associated with the game or its compatibility. The game may also be upgraded, that is, additional content may be created and released to enhance the original game in order to, for example, extend the life of the game. (Bates 2001, p. 218)

5.3.3 Game development processes and growing complexity

As was discussed above, the game development processes are increasingly growing in size and complexity along which the requirements for project management are changing dramatically (Walton 1998; vom Scheidt 2000). As in software development in general, (Schmidt, Lyytinen et al. 2001) in game development it is commonly agreed that there are more projects that are cancelled, slip or don't reach the desired quality than projects which are completed on schedule, budget and with the desired quality (vom Scheidt 2000). As for considering the game development similar to the production of any other entertainment products such as movies, this kind of comparisons have been criticized for being misleading by arguing that game development is essentially a software development project and that the techniques to be applied to manage game development processes should hence be taken from the realm of software engineering (Bates 2001, p. 220).

Whereas the academic research on the game development processes and the related problematics is, for the time being, practically non-existing, some suggestions to curtail the problems in game development processes can be found in game development practitioner literature. First, neither waterfall nor code-and-fix is considered particularly suitable for, although commonly used in, game development (Ryan 1999b). As for waterfall model, this is considered too strict for game development because building a game is essentially a process of constantly adjusting the design based on what has been built so far (Bates 2001, p. 226). Moreover, design adjustments are crucial in situations where rival games come into market during the development and the design has to be changed in order to protect the competitiveness of the game under development (Ryan 1999b). As for the code-and-fix, in turn, the problems of this approach are clearly recognized in game development (Walton 1998; Ryan 1999b; Muzyka 2000). Interestingly, as a solution a balance of the waterfall and code-and-fix types of approaches, that is, controlling chaos by planning on iteration and

change is recommended in the game development literature (Ryan 1999b). Also recommended is the use of different process models in different portions of the game development process (Bates 2001, p. 230).

Second, to support controlled iteration in game development process, comprehensive change control is recommended. This is needed to embrace the game vision and accept only the relevant changes, and requires that the original game design, the people involved, the content of the change requests, and the relative importance of schedule, budget, and product features are thoroughly understood (Crabtree 2000). Ideally, alternative ways to implement the requested changes and impact of each potential change to schedule, budget and quality should be analyzed (Crabtree 2000; Bates 2001, p. 225). Also, the change procedures, that is, how the requests are made, processed, and approved, should be defined (Crabtree 2000).

Third, modifiability is emphasized also in game development projects (Muzyka 2000). In addition to supporting iteration, modifiability plays an important role in maintenance, which is, today, needed even in games (Walton 1998). Fourth, the importance of documentation is clearly acknowledged as important in game development literature (Walton 1998; Ryan 1999a; Ryan 1999b; Davis 2000; Muzyka 2000). The documentation, for example, helps exchanging ideas and reduces wasted effort and confusion and makes it possible for the various game development parties to share the same vision of the game (Ryan 1999a; Davis 2000). Finally, use of prototyping and demos is recommended to avoid expensive redos (Muzyka 2000). Prototypes can be used to communicate the game vision, answer technical questions, debug design ideas, and test the production environment, methods and tools so that the actual product can be optimally developed (Oster 2000).

6 RESEARCH FRAMEWORK

The empirical part of this study explores game development processes in a Finnish mobile game development company. The specific research questions addressed are "what are the game development processes like in terms of high-level process models?" and "what are the needs of the game development processes in terms of support areas?". Answering to these questions is based on the premise that different software processes differ in their needs, whereas different high-level process models differ in their capabilities to support these needs. Consequently, different types of software processes implement and/or need to be supported with different types of high-level process models. This idea and the generic areas in which high-level process models differ from each other and in which different software development processes may have different needs have been presented in table 2 and in figures 15 and 16 in section 4.4.

In fact, as illustrated in figure 18, the figures 15 and 16 can be composed into a framework that can be used to analyze software processes. First, the framework can be used to analyze the needs of forthcoming software processes, and thus, to support finding an appropriate high-level process model to meet these needs. Second, the framework can be used to retrospectively analyze past software processes in terms of high-level process models and support areas.

To be more detailed, the framework can be applied for software process analysis as follows. The figure 15, supports software process analysis by drawing the attention to the areas of requirements elicitation and evolution, software modifiability and clarity, progress visibility, and risk management. Analyzing the role of these areas in the software process under examination helps understand the support needs of the process, and thus to answer the question "what are the needs of the software process in terms of support areas?". Using the figure 16, in turn, the software process under examination can be positioned with regard the different types of iterativeness versus linearity (learning versus manufacturing orientation). Further, the software process can be positioned with regard the number and nature (internal versus to customer) of deliveries and early versus late progress visibility in form of operational software. The positioning of the software process with regard the above factors helps uncover the possible resemblance between the examined software process and the existing high-level process models and thus helps answer a question "what is the software

process like in terms of high-level process models?". (For the detailed discussion on the contents of the figure 16, see section 4.4.)

The needs of the past and/or forthcoming software process regarding the areas represented by figure 15 are reflected in the appearance of the software process, that is, in the process' position in figure 16 (for the detailed discussion on the relationships between the support areas and elements of the figure 16, see section 4.4.). Thus, analyzing the role of support areas in a given software process not only leads to an increased understanding of the examined software process but also essentially supports answering to the questions "what kind of high-level process model should be used to support the software process" (in case of forthcoming software processes) and "why did the software process implement a specific high-level process model?" (in case of past software processes).

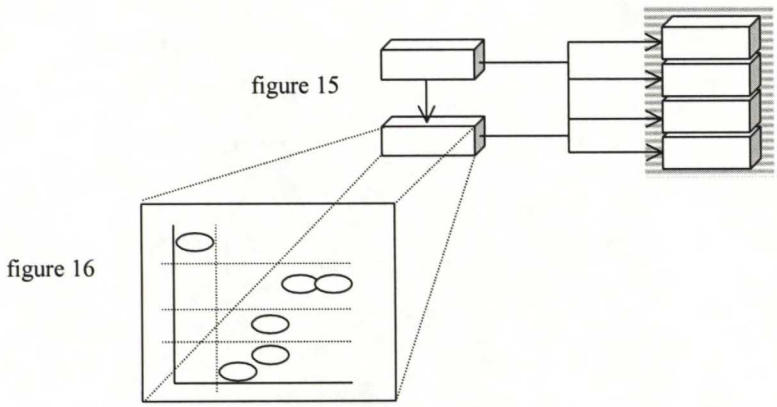


Figure 18. The figures 15 and 16 presented in the section 4.4 can together be used as a research framework when analyzing software processes.

The use of the abovementioned framework for retrospective software process analysis is demonstrated in the empirical part of this study, where the framework is used to address the research questions as follows:

- The first research question "what are the game development processes like in terms of high-level process models?" is answered by positioning the described game development process with regard the elements presented in the framework and illustrated with the figure 16. As the described software process is not directly compared to the existing high-level process models but analyzed with a framework for positioning different high-level process models, the risk of forcing the described process to fit some existing high-level process model is reduced.

- The second research question "what are the needs of game development processes in terms of support areas?", then, is answered by analyzing the role of the support areas presented in the framework and illustrated with the figure 15 in the described game development process. The role of the support areas reflects the needs of the game development process in question and affects the way the process looks like in terms of high-level process models. Thus, the answers to the second research question help form an increased understanding of the examined phenomenon and essentially support answering to the first research question.

7 RESEARCH METHODOLOGY

This chapter discusses the research methodology applied in the empirical part of this study. As the theoretical base for high-level process models is incoherent and moreover, the research on game development processes is practically nonexistent, the questions addressed in this study are of an exploratory nature. The aim is to conduct a pilot study in the area of mobile game development and thereby begin forming a basis for further game development research. Thus, only a sort of introductory overview of the phenomenon, that is, game development as a software process, is aspired in this study.

Due to the abovementioned explorative aim and lack of strong theoretical base, a case study methodology was chosen as a research method for the empirical part of this study. Further reasons for choosing case study methodology were the focus on a complex non-historical phenomenon, and the lack of need to control the actual behavioral events in the processes researched. (Benbasat, Goldstein et al. 1987; Yin 1990, pp. 13- 17)

For the purposes of this study, the number of case companies is limited to one, Codetoys, Inc. This is because the case study now conducted is intended as an exploratory device that acts as prelude to further studies on game development processes. Further, although the case study is now limited to only one company, an overview picture of two game development processes at the company in question is given.

The data collection was conducted through two interviews. Interviews are considered an essential source of case study evidence as well-informed informants can provide important insights into a situation especially if complex issues are being studied (Yin 1990, pp. 89-91; Hannabuss 1996). As the research in game development is at its explorative phase and thus, no profound understanding of the phenomenon has been yet reached, the insights of informants can prove extremely valuable. Further, as software processes in general can be considered a very complex issue, interviews were considered an ideal data collection method for the purposes of this study.

The informant was selected so that an adequate retrospective overview of at least one game development project and the related software process can be composed. Accordingly, the informant interviewed was a software engineer having both experience in game development

and the substantial operational responsibility in a number of game development projects at the case company. After it became clear that the informant can give the adequate information on more than one game development projects, the interviews were limited to one informant.

The first interview was conducted in late January 2003. This interview was of a semi-structured, in-depth nature. In order to ascertain that the informant can give an answer to all the important questions he was provided with the general topics of the interview well beforehand (see appendix 1). Also, an interview guide (see appendix 2) was prepared and used by the interviewer to ensure that none of the main themes are left without attention. The themes to be discussed in this interview and to be included into the interview guide were derived from the research questions and selected so as to bring forth the phasing, iteration, incrementality, requirements and design specification and evolution, documentation practices, use of prototypes, need for modifiability and progress visibility, risk and project management practices, reuse, and the use of high-level process models in the game development processes in question.

This one-and-half hour interview was tape-recorded in order to avoid possibility of memory lapses and to support correctness of interpretation. In order to help organize and group the large amount of chaotic interview data into clear process descriptions, an adaptation of affinity diagram method (also known as KJ method), a method often used to support, for example, process improvement efforts (see, for example, Anjard 1995; He, Staples et al. 1996; Ohiwa, Takeda et al. 1997), was applied. Affinity diagram is, essentially, a visual tool that can be used to organize seemingly piecemeal items of verbal data into themes. In executing the affinity diagram method, the different ideas and elements were first elicited from the interview data and written down. The ideas and elements were then gradually arranged into related hierarchical groups, starting from the broad categories and proceeding gradually to more refined groups. The groups were then named (given a header) so as to best describe the theme of each group. After the relationships between the groups had been examined, the structures of the processes described by the interviewee were clearly visible. The resulting process description was then gradually interpreted and shaped into more literary "essay" form. Also, preliminary process charts were prepared to illustrate the flow of events described by the informant.

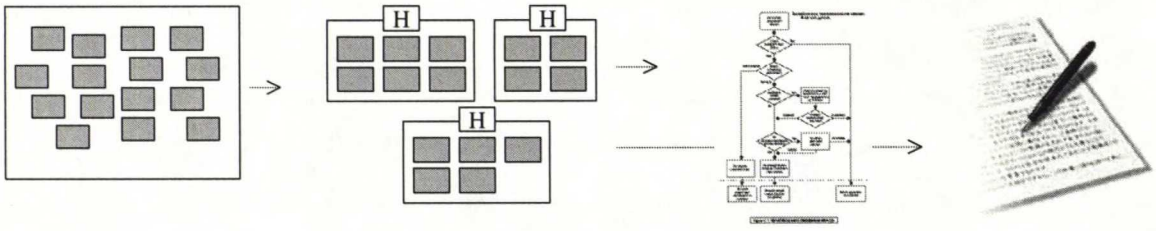


Figure 19. Affinity diagram was used in this study as a visual aid to organize seemingly piecemeal items of verbal data into related themes and then, finally, into process charts and text form.

The results (essay form descriptions of the processes and the process charts) of the preliminary analysis based on the first interview round were corroborated by the second, more structured interview conducted in the early February 2003. This second round was carried out via email and its goal was to verify that the interviewer had understood the informant correctly but also to complement the results of the first interview. Accordingly, in this interview the informant interviewed in the first round commented the results of the first interview round but also answered to some new, more refined questions.

The data received through the second round was organized to complement and adjust the process descriptions and charts prepared based on the first interview round. After this the process descriptions and charts were analyzed with the help of research framework as described in chapter six of this study. The resulting findings of the case study along with the descriptions and the framework-based analysis of the two game development processes described by the informant are presented in chapter eight.

8 CASE CODETOYS – ZOBMONDO!!

This chapter discusses the case study in which two software processes related to the development of Zobmondo!! mobile game at Codetoys, Inc. are examined. The objective is to explore these two game development processes with the help of framework presented in the chapter six in order to answer the following questions "what are game development processes like in terms of high-level process models?" and "what are the needs of game development processes in terms of support areas?".

The chapter is divided into five parts. Section 8.1 presents the organizational context for the study and section 8.2 provides some background for the game Zobmondo!!. In sections 8.3 and 8.4 the software processes under examination are described and analyzed. Finally, in section 8.5 the findings of the case study are discussed.

8.1 Organizational context

Codetoys is a globally operating Finnish company providing mobile entertainment services (see figure 20). Codetoys' customers and partners include telecommunications operators, handset manufacturers, media portals and companies, and brand and content owners. Codetoys has established customer relationships with over 50 mobile operators in more than 40 countries, making its games available to over 250 million consumers across all continents and therefore, the most widely available mobile games in the world. The company also has the industry's most complete set of mobile games and services based on globally recognized brands. Codetoys has licensed wireless rights and released the mobile versions of brands such as Who Wants To Be A Millionaire?, Trivial Pursuit, Zobmondo!!, E.T. The Extra-Terrestrial, and Subbuteo. The product portfolio includes also private label games like Tease, Mobilization, and Virtual Swing, that can be launched under the customer's own brand.

Codetoys has development capabilities covering SMS, MMS, J2ME, SymbianOS, WAP and i-mode. In addition, the company has actively participated in development of an industry-standard platform for wireless entertainment applications with Mobile Games Interoperability Forum (MGIF), an industry forum founded by Ericsson, Motorola, Nokia and Siemens. In 2002 Codetoys was awarded the Finnish INNOFINLAND Award for its interactive mobile services platform, which enables the design, development and global distribution of wireless

entertainment applications with easy adaptability to meet the needs of different markets. The same year company also received a GSM association award for "best consumer wireless application" for the mobile game Who Wants To Be A Millionaire?.

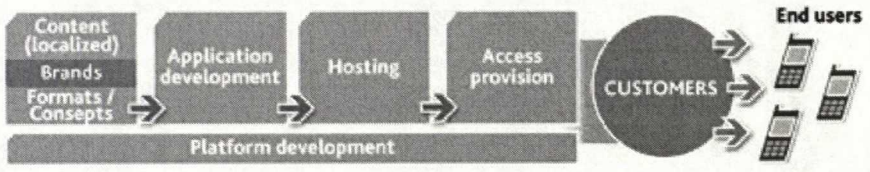


Figure 20. Entertainment services provided by Codetoys cover the whole value chain from application and platform development to technical integration and support.

Codetoys was formed in March 2002 by the two leading Finnish mobile entertainment companies, Codeonline and Springtoys. Founded in 1998, the company is headquartered in Finland, with offices in the UK, Germany, the US and Japan. Investors include AOL Time Warner Ventures, Bertelsmann Capital Ventures and Motorola Ventures. Currently the company employs 23 persons.

8.2 Background for the game Zobmondo!!

The initial development of the mobile game Zobmondo!! was started at Codetoys (back then Codeonline) in the fall 2000. The concept for the Zobmondo!! originated from a popular American board game published by game giant Hasbro and invented by the UCLA student Randy Horn. Zobmondo!! is a trivia type of social game in which the players take turns asking and answering bizarre "would you rather..." questions (for general information on Zobmondo, see http://www.zobmondo.com and appendix 3). The Codetoys, then, had an intention to enter American markets by developing a mobile phone version of this popular board game. In mobile phone version of Zobmondo!!, which, for that matter, can be considered to represent the casual games genre discussed in section 5.2, the player answers the abovementioned type of questions but also tries to guess the opinion split of the other players responding to the same questions. Based on the accuracy of the guesses and the answering speed, the players can then be ranked and, for example, a prize can be awarded to the players with the highest scores. The mobile phone versions of Zobmondo!! developed by Codetoys have been launched at least in the UK, Finland, South Africa, and the United States. All in all, this game is now available for mobile phones in text messaging (SMS), mobile Internet (WAP and HDML) and J2ME versions.

Figure 21. Zobmondo!! is available in board game, mobile game and TV-show formats.

As was already discussed above, the Codetoys' original intention was to enter US markets by developing a mobile phone version for young people of the already successful American board game. The game development began with Java and HDML (kind of early WAP used in the United States) versions of the product, but as the release of the phone for which the Java version was intended was heavily delayed, the development of Java versions was terminated unfinished. The development of the HDML version of the product was, however, continued and after about three months of the launch of the HDML project, the HDML version was introduced in mobile entertainment exhibition in February 2001. After this the negotiations with the US telecommunications operators begun but, unfortunately, did not succeed. The novelty of the technology and communication infrastructure caused problems for the US operators, which, therefore, were not interested in investing to new products and services.

Consequently, the fully operational HDML version published in the exhibition was never moved into production. It was, however, used for demonstration purposes and as a basis for the later WAP and "pruned" SMS versions of the product. As the US operators later on turned down also the WAP version of the product, the game was published in AT&T portal to elicit user feedback through "sink or swim contest" where the new games were tested by and fought for the attention of real users. Further, as the concept had not, so far, actually brought any money in, the SMS version of the product was published in Finland in games-focused television show "TILT" and was also later sold to various operators.

In this case study, we first discuss the software process related to the development of the initial HDML version of Zobmondo!! and then continue by exploring the generic software process related to the operator-specific tailoring of the Zobmondo!! SMS version.

8.3 Zobmondo!! HDML version

The section 8.3.1 describes the development of the HDML version of Zobmondo!!. The description is divided into phases of generic game development process discussed earlier (in section 5.3.2). The section 8.3.2 then analyzes the HDML software process in terms of high-level process models and support areas with the help of framework presented in the chapter six.

8.3.1 Description of the HDML software process

The development of HDML version of Zobmondo!! was started in December 2000. The development schedule was dictated by the need to publish a fully operational product at a mobile entertainment exhibition in February 2001 and the will of the license owner to see quick results. As for project management, no formal project plan was prepared. The requirements engineering was done as teamwork but the subsequent design and implementation activities related to the game logic module were a responsibility of one developer only. Testing, then, was conducted by a specialized department.

Concept development: The development of Zobmondo!! HDML version was started with a concept development phase. As for the selection of this particular concept, Zobmondo!!, one thing affecting this was that Codetoys had already successfully produced one trivia game, "Trivial Pursuit", for mobile phones and was at the time developing a mobile phone version of the famous "Who wants to be a millionaire" trivia. At the time, Codetoys was focusing on trivia type of games as mobile gaming was only in its infancy and trivia games were one of the few game types making sense in SMS, HDML and WAP formats. Further, the experience and tools necessary for development of this type of games were already in place and acquired. Moreover, development of a platform providing the basic functionalities for trivia type of games had been carried out at Codetoys and this platform could be utilized in the development of Zobmondo!!. Consequently, development of only a relatively small game logic module was required for Zobmondo!! This separation of the platform development and game logic development made the development related to concept Zobmondo!! somewhat smaller and less riskier effort than it otherwise would have been. Other criteria considered when assessing the feasibility of the concept Zobmondo!! were the novelty and interestingness of the concept, and naturally the payment and other conditions applying to the use of the Zobmondo!! license that had to be acquired from outside.

Together with the inventor (which also owned the license for the game) of the original Zobmondo!! board game the representatives from different business functions at Codetoys carried out the early concept development phase. Different possibilities for the "mobile manifestation" of the game were assessed and mapped out and rough mock-ups representing the mobile phone display were prepared to experiment with different possibilities for the flow of the game and to get the early taste of the feel of the game. The goal was to follow the original board game concept as closely as possible and find a way that would maintain its spirit and idea to the highest possible degree. Understandably, mobile phones with the HDML format, as the new medium for Zobmondo!!, forced the development team to make some allowances and changes.

Pre-production: After the original board game concept had been adjusted to fit into the mobile space the specification of more detailed requirements for the Zobmondo!! HDML version was started. The use-case methodology was applied in order to assess the possible scenarios and sequences of interactions between the game and the users, and to clarify and organize the system requirements. The usability and playability of the game along the limitations set by the HDML format and mobile phones in general were taken into consideration and end user feedback was elicited through inquiries and by letting the end users test and comment the early throw-away prototypes of the product. The requirements specification was documented and used as a guide later in the development process. The requirements stated at this point of time applied, as such, pretty well for the early HDML version of the Zobmondo!! but were thoroughly revised and adjusted during the development of, for example, later SMS version that was based on the HDML version now discussed.

As a separate platform was used to provide the basic functionalities for the game, a decision had to be made as to which version of the platform is used and the requirements set by the platform had to be taken into consideration during the game logic module related design activities. Operational prototype was used to support the design activities and modularity was striven for to make the quick development of the product possible. Technical documentation for the platform was supplemented with the one prepared for the game logic module to act as a guide later in the development process.

Implementation and the latter testing phases: The HDML software process quickly proceeded to code and test activities, because of the tight schedule. During this phase the architectural

prototype earlier developed was fleshed out with the remaining components and application features to achieve the full product. The coding was done using Java programming language and Java-coding standards were applied to guide the work. The implementation was carried out in a number of iteration rounds that consisted of incorporation of new components and fixing and adjusting of the functionality provided by the existing components (see figure 22). Each (main) iteration round resulted in a build, that is, kind of pre-release version of the product. Inside the iteration rounds were smaller iterations related to the coding, testing, fixing, and integration of smaller components. Configuration management practices were used to keep track of the composition of different intermediate versions and the differences between them. As for testing, only internal testing was carried out for the HDML version as the version was developed for demonstration purposes only and was not intended to be used by customers as such.

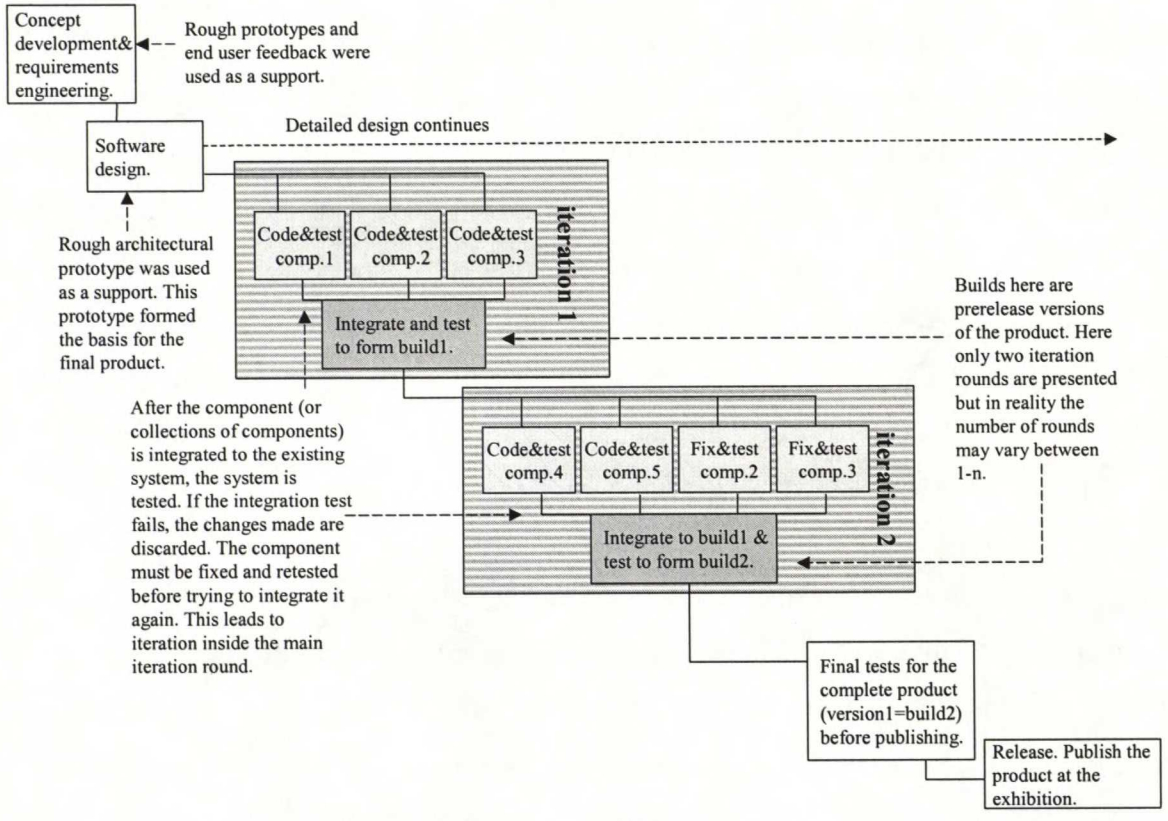


Figure 22. Zobmondo!! HDML software process described by the interviewee.

During implementation iterations the detailed software design evolved somewhat. The technical specification was updated from time to time, as the tight schedule and iterative nature of the process did not make the continuous updating activities feasible. This did not, however, cause problems because after the requirements engineering phase, the development

of the game module was (except for some parts of testing) mainly the responsibility of one person only and moreover, the game logic module was relatively small in size. If an up to date picture of the design was to be formed this was mainly done by reading the source code. The technical documentation was, however, thoroughly brought up to date at the end of the project.

Release and Evolution: The HDML version was successfully published at the mobile entertainment exhibition in February 2001, about three months after the project launch, and was used as a basis for, unfortunately not so successful, negotiations with the US operators and the later WAP and SMS versions of the product.

8.3.2 Analysis of the HDML software process

Process in terms of high-level process models

No decision was made to use any specific high-level process model in HDML software process. The HDML software process was, however, clearly of planning-oriented, incremental nature. The requirements were thoroughly specified at the beginning of the process (using even operational throw-away prototypes) and acted as a guide for and remained stable during the rest of the development. After requirements engineering the software design activities were conducted. High-level design was supported with architectural prototype, which then, during implementation was fleshed out with the remaining components and application features to achieve the full product.

After the high-level design, the HDML software process was carried out incrementally in iteration rounds that were planned beforehand and consisted of detailed design, coding, testing, and integration related to the addition of new components and fixing of existing components. Each of the iteration rounds was related to the implementation of specific requirements and resulted in an intermediate executable version of the product (increment being the difference between the results of successive iteration rounds). Implementation was based on requirements and technical specifications earlier prepared, and the testing of and experimentation with the intermediate product versions was carried out to find out possible problems and errors, rather than to build more thorough understanding of the system under development. Thus, the purpose of the incremental approach in Zobmondo!! HDML version

process was not so much about experimentation or learning as about gradual assessment of the success of implementation that was based on thorough planning.

Based on the above, it can be concluded that Zobmondo!! HDML software process with its incremental (multiple deliveries) and internal nature combined with planning and control orientation clearly resembles incremental development rather than, for example, the more experimentation-oriented increment-based models of iterative enhancement and evolutionary delivery. As opposed to incremental development, however, the division of the development into iteration rounds and thus, into increments, was not as thoroughly planned or systematic as model of incremental development promotes. Accordingly, the approach used in the development of Zobmondo!! HDML version is in the figure 23 represented little bit right to the incremental development, implying that the HDML software process was of somewhat less controlled nature.

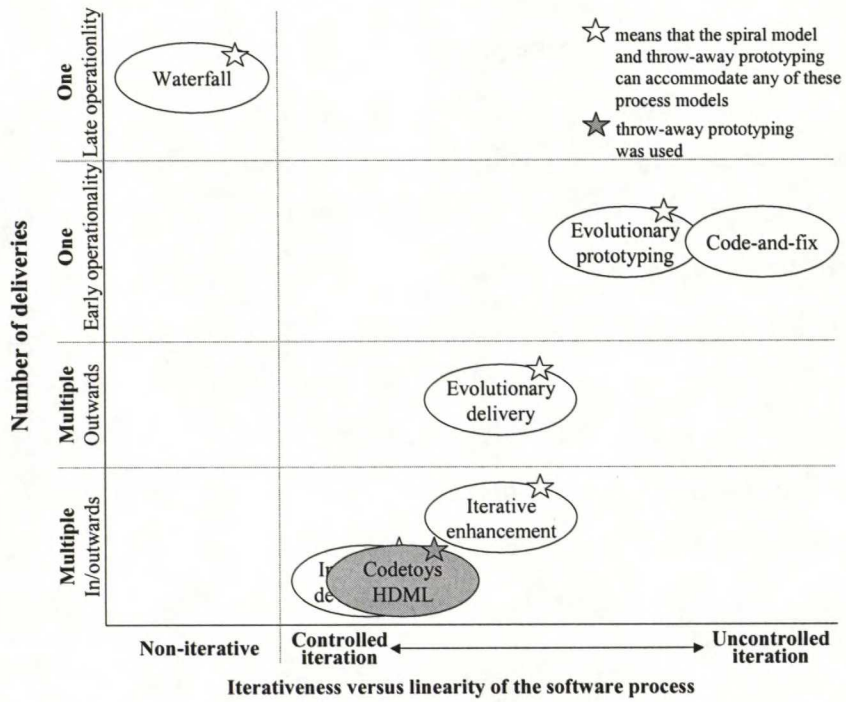


Figure 23. Positioning Zobmondo!! HDML software process based on analysis.

Process in terms of support areas

Requirements elicitation and evolution: HDML software process was about development of a fully operational but preliminary version of a product that, as a concept, was new. Consequently, the process's capability to support concept development and the related

requirements elicitation was important. The requirements were thoroughly planned in the beginning through experimentation with both operational and mock-up throw-away prototypes. Also feedback from real end users was elicited. After this, the development was quite straightforward and was conducted based on a stable requirements specification. To conclude, requirements elicitation formed the basis for the software process and was thoroughly conducted at the beginning. Requirements evolution did not really take place.

Software modifiability and clarity: Software modifiability and clarity were incorporated into process as its basic elements. Further, the design activities were systematically conducted and configuration management applied. The technical documentation was prepared before implementation but was not continuously kept up to date during the quick-paced development. The importance of comprehensive up to date technical (and other) documentation was, however, clearly acknowledged as this was considered one of the end results of a software development project. To sum up, software modifiability and clarity were considered important in HDML software process.

Progress visibility: Internal progress visibility in HDML software process was provided by incremental development approach and early operational versions of the product. Whereas technical documentation was not continuously kept up to date, the stable requirements documentation could be used as a yardstick to assess the product's degree of completeness. Further, no formal project plan was prepared to track the progress. As for external progress visibility, this was not really needed because of internal nature of the project. To summarize, progress visibility in other forms than operational software early and often was not really considered important in the HDML version development. Proceeding to implementation quickly was emphasized as publishing the game as planned was paramount for the company.

Risk management: Risk consideration and management seemed not to play an important role at the project level in the development of HDML version. However, the risks related to the concept Zobmondo!! in general were to some degree considered in the early concept development phase.

8.4 Zobmondo!! SMS version

Section 8.4.1 discusses briefly the development of the preliminary SMS version of Zobmondo!!. As the preliminary SMS version was, basically, developed pruning the HDML version discussed above, no thorough description of its development is here given. Section 8.4.2, then, discusses the generic process of tailoring the preliminary SMS version in various operator-specific projects to achieve operator-specific SMS versions of the product. Finally, in section 8.4.3, this generic operator-specific SMS software process is analyzed in terms of high-level process models and support areas with the help of framework presented in the chapter six.

8.4.1 Introduction

As was already discussed above, after the unsuccessful negotiations with the US operators regarding the Zobmondo!! HDML version, WAP and SMS versions of the game were developed. Whereas the WAP version was a refined version of the earlier HDML version, the SMS version was, basically, developed by pruning the HDML version. Consequently, the software process for the preliminary SMS version was lighter than that of the original HDML version as the development did not have to be started from scratch. Just like the development of the HDML version, the development of the SMS version was started "without customers" with the intention to use the resulting preliminary version as a basis for customer negotiations. The mobile Internet versions (HDML and WAP), and SMS versions of the product were all developed in one year.

The preliminary SMS version of the product was developed based on generic requirements and design specifications, which then, in customer-specific projects, evolved to customer-specific specifications based on customer feedback. In other words, the generic, preliminary version used as a basis for negotiation was gradually modified into a tailored version (or variation) through iteration based on customer feedback.

In every customer-specific SMS project, some level of tailoring was needed to make the product meet the needs and vision of the customer. The original goal was to carry out the customer-specific tailoring mainly through parameterization using configuration files made for this purpose, and this way to considerably speed up the customer-specific development (To put it simply, the configuration files were files expressing the possible range of values for

a given attribute with each value resulting in a different outcome when chosen. Parameterization, then, is choosing the most appropriate value for the attribute for a given purpose). In reality, however, only part of the tailoring could be accomplished this way and the rest of the changes still needed to be done the old-fashioned "coding way".

A generic description of the flow of the events in the customer-specific SMS software processes now follows (customers being operators and the process hereinafter referred to as operator-specific SMS process).

8.4.2 Generic description of the operator-specific SMS software process

To begin with, the operators were approached with an operational and quite refined preliminary version of the product. If the demonstration got the customer interested the negotiations continued with requirements specification activities, which normally were carried out as quickly as possible so that the implementation of the changes could begin early. The operator-specific requirements were elicited by further demonstrating the game to operators and by letting them experiment with the preliminary version into which the basic playability and functionality had already been incorporated. Sometimes also simple throw-away prototypes not using the real platform were used to support requirements elicitation. Based on the experimentation and demonstrations the customers and developers were able to identify the areas needing changes and further development. These were then documented and shaped into a form of change requests to guide the development.

In each operator-specific SMS process the preliminary version used as a basis for negotiations evolved to a different operator-specific version or variation of the product. If the customer did not propose major changes a new variation of the preliminary version was developed. On the other hand, if the changes proposed were significant, these were implemented in a new version (revision) of the product (see figure 24). Normally, the changes proposed by the customers were about the appearance of the game and suchlike, and could often be implemented through configuration files. The bigger issues concerning the game concept and game logic were always left untouched by operators as this would have required thorough renegotiations with the license owner.

Operator-specific parameterization was done by a specialized team. Naturally some of the changes could not be implemented through parameterization, but required adding of new components and adjustment of the existing ones. In addition to these, the need for coding in operator-specific SMS process was due to need to fix the errors found in existing components. Like in the development of the HDML version, the coding was done in Java programming language and Java coding standards were used to guide the work. As for the software design, this was intentionally made so generic that normally only little modifications to the detailed design were needed in operator-specific projects.

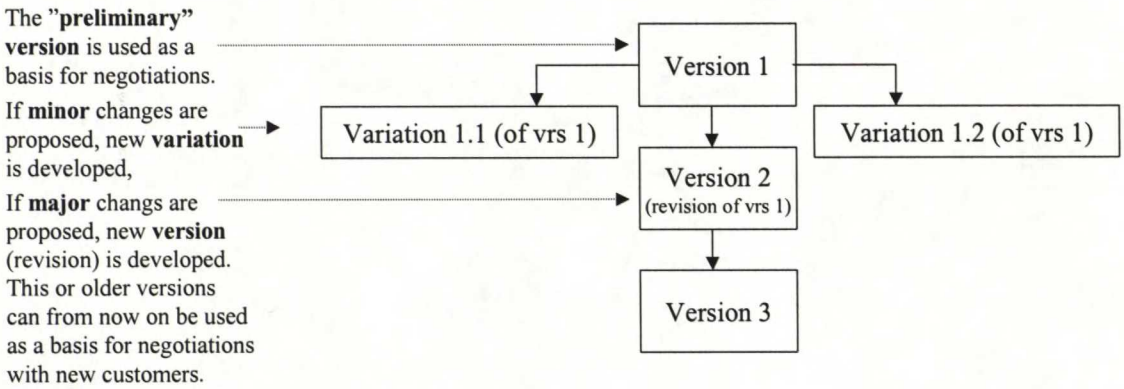


Figure 24. The variations form the branches whereas the new versions (revisions) lengthen the trunk (main branch) of the version tree.

The required addition of new components and modifying of existing components was carried out incrementally in a number (1 – n) of iteration rounds (see figure 25) each of which resulted in an internal release of the product. Through each iteration round, new components could be incorporated, and the functionality of the existing components adjusted and errors fixed. Configuration management practices were used to keep track of the composition of different intermediate versions and the differences between them.

After the required changes had been successfully incorporated into the product and the product had passed the internal testing (for example, system, performance and stress testing), the product was delivered to customer to be either accepted as such for beta testing or to accepted only after further changes and iterations. Often, this resulted in new change requests being made as the preliminary requirements proposed by the customer usually evolved as the development proceeded, sometimes even to a great degree. The operators seemed to have problems deciding what they really want and sometimes the minds were changing weekly. At times this was problematic as "however unstable, fluctuating, or even irrational, the changes proposed by the powerful operators had to be implemented". Further, one thing in common to

all operator-specific projects seemed to be that the thorough requirements specification at the beginning of the project was thrown out of the window as the development team rushed to the implementation of the proposed changes- "in retrospective, the initial requirements specification was never done as thoroughly as it should have been done".

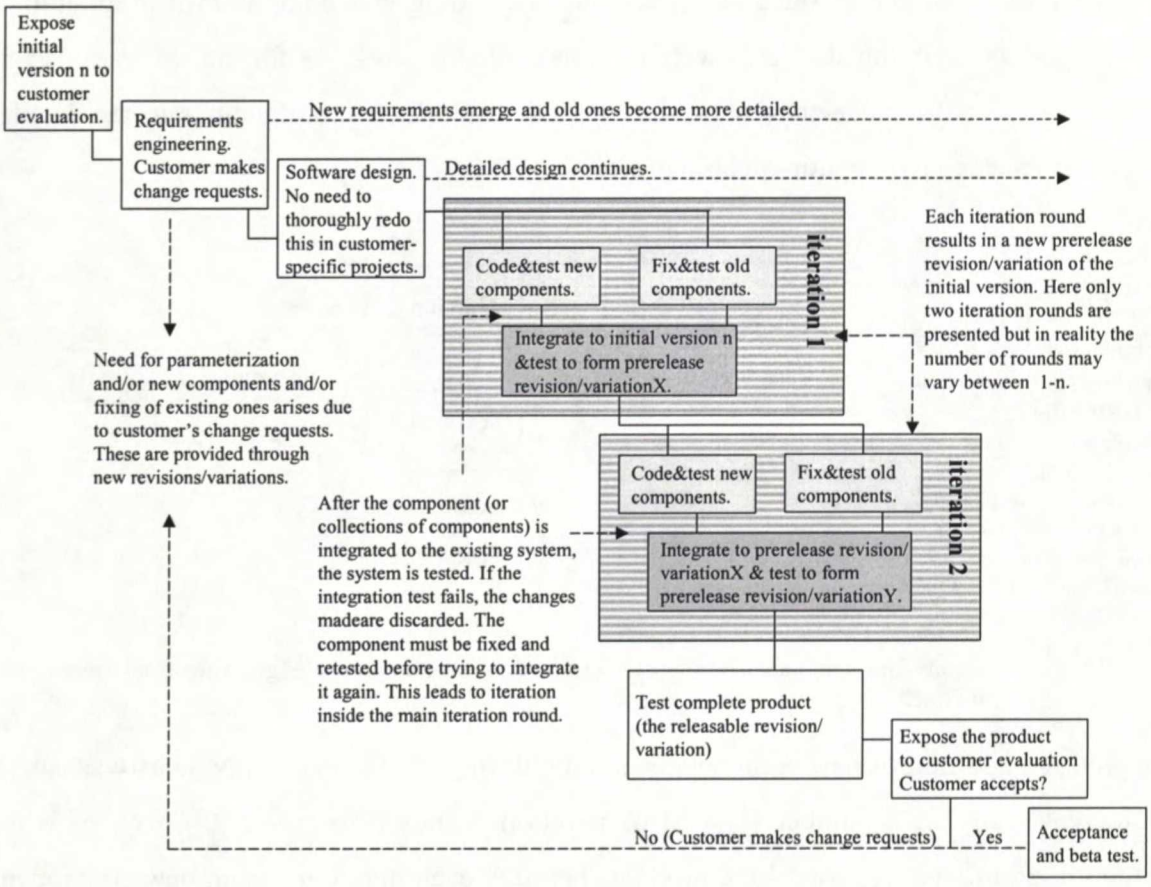


Figure 25. Zobmondo!! operator-specific SMS software process described by the interviewee.

The iteration "evaluate-modify" was terminated only after the operator had accepted the product for beta testing. As for the beta testing, this sometimes resulted in new previously unnoticed faults in the software being uncovered and thus, new iteration rounds were needed. When the product finally passed the beta testing and was ready for being tested by the real end users, "a soft launch" was first made. Soft launch was about publishing the game without promotion so that initial end user reactions and feedback could be gradually elicited. Moreover, soft launch gave the opportunity to comprehensively test the product in real use and fix faults that had not earlier emerged. In fact, finding all the possible faults before publishing was not even striven for, because this would have delayed the publishing of the game excessively. After the publishing the product was maintained by mainly fixing new faults but also by delivering some new content (new questions) for the game.

As for project management in operator-specific SMS projects, a formal project plan was always made at the beginning of the project. Also a "launch schedule" was prepared and a decision made as to when the tailored product is delivered to customer for beta testing. The operator-specific projects were very organic and involved a lot of cooperation with the customer, especially towards the end of the project when the final adjustments were being made.

As to documentation, the guidelines had been established as to which documents should be made for each project and the projects were not terminated before the required documentation was in place. Some documentation was always done to guide the development activities but as the products had to be developed quickly and iteratively, the detailed updating of the documentation was left to be done at the end of the project.

8.4.3 Analysis of the operator-specific SMS software process

Process in terms of high-level process models

Also operator-specific SMS processes were conducted without a decision to apply some specific high-level process model. Based on the description given by the interviewee, however, it can be concluded that operator-specific SMS processes greatly resembled evolutionary prototyping. A preliminary operational implementation (sometimes along with throw-away prototypes) representing the entire solution was shown to operators to let them experiment with the product so that customer feedback could be elicited. As a result of the feedback, the requirements specification was updated and the detailed design modified. After the changes proposed had been incorporated, the entire solution (as opposed to increment or partial solution) was again exposed to customer evaluation. This, then, resulted either in further iteration or acceptance (delivery). Thus, the operator-specific SMS process was essentially an experimentation-oriented process.

On the other hand, SMS process was much too controlled to be considered "pure" evolutionary prototyping. As opposed to the pure evolutionary prototyping, both the importance of comprehensive technical specifications, planning and modifiability were emphasized. Moreover, although documentation was not up to date all the time, its importance was clearly emphasized and was considered as one of the project end-results.

Thus, in a way, the approach used in SMS process could be seen as an advanced evolutionary prototyping model where early operationality, iteration with the entire solution, and one delivery consisting of the entire solution is combined with the iteration of more controlled nature.

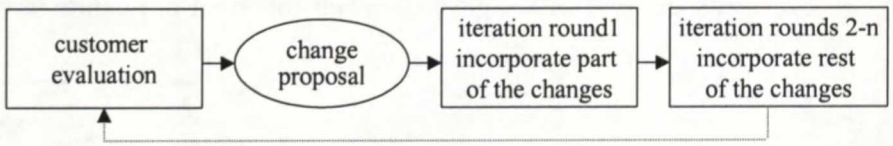


Figure 26. Only the result of the last iteration round is exposed to customer evaluation. Although the solution may have been built in increments, the customer evaluates entire solution.

As for the incorporation of changes requested by the operator, from the developer point of view this was done incrementally in a number (1 – n) of iteration rounds each of which resulted in an internal release of the product. Now the difference between the releases of successive iteration rounds can be considered as an increment: changes made during the iteration round 2 are incorporated to the result of iteration round 1, and so forth. Thus, operator-specific SMS process also has elements of increment-based process models. In operator-specific SMS process the results of the early iteration rounds were only internally delivered and used as a basis for the further iteration rounds, of which only the result of the last one was finally exposed to customer evaluation (see figure 26). Further, the purpose of the division to the iteration rounds (and thus increments) was not to so much to learn more about the product in each iteration round, as to only gradually evaluate the success of the changes incorporated. Consequently, incremental development is the one of the increment-based approaches this process resembles the most. However, the division of the development into iteration rounds and thus to increments was not as thoroughly planned and systematic than incremental development promotes, implying that the approach used was of less controlled nature.

To conclude, the operator-specific SMS process resembled evolutionary prototyping, but the iteration in the process was clearly of more controlled nature. From the developer point of view the changes requested by the operator could be incorporated to the product incrementally before exposing the product to customer evaluation again, and thus, also some elements of the incremental development can be seen in operator-specific SMS process. However, as the iteration and thereby requirements and design evolution in operator-specific SMS process applied to entire product, we consider the process to represent an approach of one delivery

rather than that of multiple deliveries. Accordingly, in the figure 27 the operator-specific SMS process is shown left to the evolutionary prototyping.

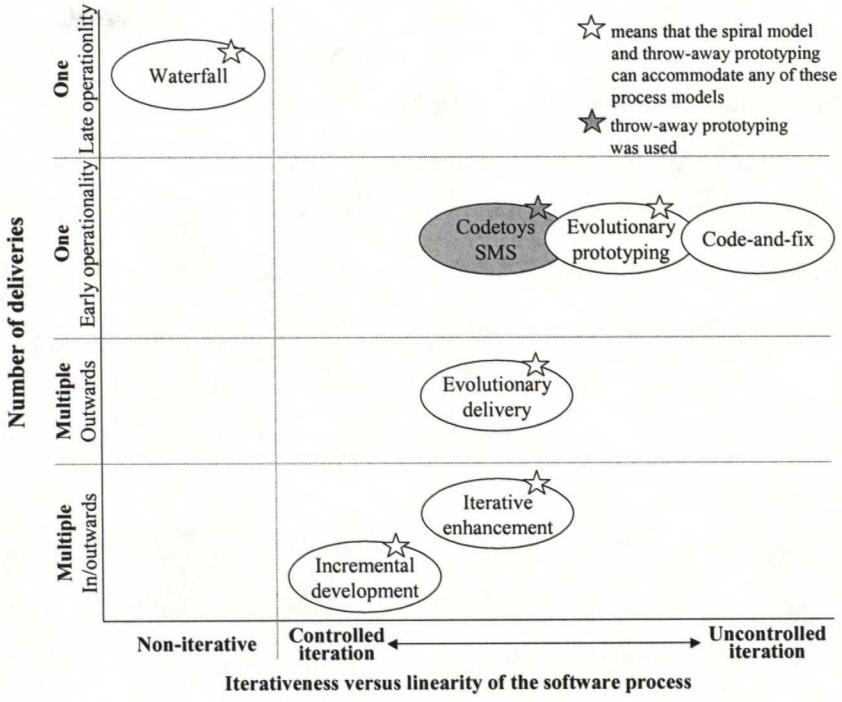


Figure 27. Positioning Zobmondo!! operator-specific SMS software process based on analysis.

Process in terms of support areas

Requirements elicitation and evolution: Requirements elicitation and evolution played a central role in operator-specific SMS process. Experimentation with and demonstration of an operational version of the product was used as a basis for sales negotiations and proved important in elicitation of customer feedback and requirements. Further, multiple iteration and experimentation rounds were needed to meet the customer requirements that both changed and got more detailed during the software process. Also, eliciting and meeting the customer requirements was vital for the developer company because of the strong bargaining position of the customers.

Software modifiability and clarity: Software modularity and clarity were incorporated into process as its basic elements. Further, systematic software design activities, and configuration management were emphasized in operator-specific SMS process. The technical documentation was prepared before implementation but not continuously kept up to date during the quick-paced development. The importance of comprehensive up to date technical

(and other) documentation was, however, clearly acknowledged as this was considered one of the end results of a software development project. To sum up, software modifiability and clarity were considered important in SMS software process.

Progress visibility: The importance of operational software early in the process was emphasized as customer experimentation with the operational software formed the basis for sales negotiations. Instead of incremental releases (to customer), that would not have been a reasonable solution for projects of this size, setting tight schedules and short time spans for the customer-specific processes were used to assure customers about project progress. Although not providing visibility for customers, incremental approach used to incorporate the requested changes naturally provided internal progress visibility. Also project plans were prepared and used, by both the customer and Codetoys, to track the project progress. As for technical documentation, this was not really the source of up to date picture of the project status. This was because continuous updating of the documentation would not have been feasible due to need for quick iteration. Requirements specification, however, was always updated based on customer experimentation and thus provided a form of status information between iteration rounds and could be used as a yardstick to assess the product's degree of completeness. To sum up, the progress visibility, especially in form operational software early and often was important.

Risk management: Risk consideration and management seemed not to play an important role at the level of single customer-specific projects. If risks were discussed this was done between general management and project manager. At the concept (Zobmondo!!) level risks were, however, to some degree considered in the early concept development phase.

8.5 Discussion of the findings

This case study examined software processes related to the development of HDML and operator-specific SMS versions of the Zobmondo!! mobile game at Codetoys. The two game development processes were analyzed with the framework developed in the theoretical part of this study to answer the research questions "what are the game development processes like in terms of high-level process models?" and "what are the needs of game development processes in terms of support areas?".

The first research question was answered by positioning the HDML and operator-specific SMS software processes with regard the elements presented in the framework and illustrated with figure 16. Although in neither of the processes was up-front decision made to use some specific high-level process model, both of them, when retrospectively analyzed, clearly had elements of specific high-level process models. The HDML software process applied straightforward incremental approach based on thorough planning. Operator-specific SMS process, in turn, combined some degree of planning with quick iteration applying to the entire solution and based on customer experimentation. Both processes considered the early operationality of the software important and emphasized the quick development of the product. Based on the analysis, it was concluded that the HDML software process resembled less controlled version of incremental development, whereas operator-specific SMS process resembled a more controlled version of evolutionary prototyping. It was also realized that although the resemblance was not due to up-front decision to use a specific high-level process model, it was not accidental either. Instead, the resemblance was due to both intentional gradual adaptation to but also careful anticipation of the emerging needs

The second question, then, was answered by analyzing the role of support areas (presented in the framework and illustrated with the figure 15), that is, requirements elicitation and evolution, software modifiability and clarity, progress visibility, and risk management, in Zobmondo!! HDML and SMS software processes.

Requirements elicitation, user experimentation and feedback played an important role in and formed the basis for both game development processes. Requirements evolution, then, did not take place in HDML software process but was the fundamental part of the operator-specific SMS process where meeting even the subtlest customer requirements was vital. The requirements stability in HDML software process may be partly due to the fact that the project was purely internal and aimed to a development of a preliminary version of the game not to be used as such by external customers. This meant that the requirements could be quite freely specified by the person(s) responsible for the game vision and having previous experience on the development of similar games. Moreover, the short time-span of the project probably reduced the possibility of external pressures necessitating requirements modification emerging during the project.

Both game development processes incorporated software modifiability and clarity, and emphasized systematic design activities and controlled design modifications. These were needed to ensure the software product's amenability to quick and painless iteration and customer-specific tailoring. Technical (and other) documentation was prepared before implementation but continuous updating activities were not seen feasible during the quick-paced implementation as this would have probably slowed down the development and after all, quick development of the product was vital in both HDML and SMS software processes. This did not however seem to hamper the development. First, in HDML software process the design evolved only a little due to planning-oriented and internal nature of the project. Further, in both game development processes the lack of up to date documentation was probably, to some degree, compensated by systematicness in design activities and controlled design modifications. Also the small size of the projects and products may have made the lack of up to date documentation during the development less severe. Documentation was, however, always updated at the end of the project.

As for progress visibility, in both processes the main signs of progress were considered to be provided by operational software. Further, operational software early in the process was emphasized and incremental approach applied provided some internal progress visibility. Steady documentation was not feasible in the quick-paced processes. As to using project plans to track progress, this was done in SMS software process but not in HDML software process as preparation of formal project plans may have not been considered feasible in a small sized internal project. The role of operational software as a main sign of progress may have something to do with the tight schedules and short time spans of the projects: overall, the focus seemed to be on getting tangible results in form of operational software quickly.

Interestingly, risk management at the single game development project level seemed not to be considered essential in either of the processes. This may have been because of the relatively small size of the products and short time-spans of the projects. In HDML process also the fact that the project was not a customer-project but of an experimental and internal nature may have affected the risk orientation of the process. However, in HDML process some risk exploration at the concept level was conducted as question was about a new game concept.

The above answers to the second question, then, give an indication of the reasons as to why, in particular, did the HDML and SMS software processes resemble specific high-level

process models. HDML software process was about development related to a new game concept with a tight schedule. Thus, the development was started with thorough planning but quickly proceeded to the implementation. Getting the game published as agreed with the license owner was paramount and consequently, thorough planning was combined with early operationality and increments to provide internal progress visibility. As increments were applied mainly for dividing the development into convenient chunks rather than for experimentation purposes, the development approach used resembled incremental development. However, as the division of the development into increments was not as thoroughly planned or systematic as the model of incremental development suggests the approach used in HDML software process was an adaptation of less controlled nature. In operator-specific SMS process, then, the needs for requirements evolution, successful repetitive modifications, and quick results had to be met. Consequently, an approach of controlled iteration with an adequate level of control was applied. Moreover, as the software product in question was of small size the necessary customer experimentation with product subsets would not have been very reasonable and thus, the iteration was applied to the entire solution. Accordingly, the approach used in SMS process resembled a control-oriented adaptation of evolutionary prototyping.

Apart from the above, one of the interesting findings of the study was the central role of reuse in the examined game development processes. In HDML software process the existing components were reused by using a platform module to provide the basic services for the game. In operator-specific SMS process, on the other hand, a quite refined but generic version of the product was used as a basis to develop multiple operator-specific SMS versions of the game. The examination of the HDML and SMS software processes also revealed the significant impact of the powerful customers and license owner on the determination of game development project scope and schedule. Further, it was discovered that customers were approached only after operational, although preliminary, version of the game was available to be demonstrated and experimented with. Thus the importance of operational software early in the process was clearly emphasized. Interestingly also, both processes applied the guidelines proposed in game development practitioner literature to curtail the common game development problems (see section 5.3.3). First, both processes applied controlled iteration and change control, and also emphasized modifiability. Second, prototyping was applied in HDML software process to support requirements engineering and architectural design, whereas SMS software process was essentially an adaptation of evolutionary prototyping

(using also throw-away prototypes as support). Finally, the importance of documentation was clearly acknowledged in the case company: documentation guidelines had been established and comprehensive, up to date documentation was seen as one of the software development project end results.

9 DISCUSSION AND CONCLUSION

First of the two main objectives of this study was to begin consolidating the theoretical basis for the high-level process models (also known as life cycle models) through provision of a comprehensive review on these models and by development of a preliminary framework for positioning them. In order to meet the objectives a literature review on high-level process models was first conducted. Based on the literature review the different high-level process models and their characteristic features were then discussed and the support provided by them to software processes analyzed. Further, the literature review formed the basis for a creation of a framework for positioning different high-level process models and for analyzing software processes in terms of high-level process models and support areas (requirements elicitation and evolution, software modifiability and clarity, progress visibility, risk management). This framework can be used for retrospective analysis of the past software processes, as well as to support analysis of the needs of forthcoming software processes and thus, the finding or creating an appropriate high-level process model to match these needs.

Second main objective of this study was to act as a preliminary step in investigating game development processes and thus help establish a basis for further game development research, as research on this area is for the time being, practically nonexistent. Further motivation for selecting game development as a topic was as follows. Firstly, game development is an area of software development that is strongly growing in importance. Secondly, and more importantly, it has been realized that game development is expressing the same problems than software development in general: as the software processes grow in size, a more disciplined way of work is often needed to accomplish them successfully. It has been proposed that game development processes should be managed as software processes and hence the remedies to cure the processes should be taken from the realm of software engineering. As one of the central elements in traditional software process improvement are software process definition and process modeling, these practices should probably be acknowledged also in game development: software process definition and modeling could help game development organizations develop comprehensive insight as to how game software is or should be developed.

To meet the abovementioned second objective a pilot case study in the area of game development was conducted to explore and give an overview of the game development

processes in an internationally recognized Finnish mobile game development company. The two game development processes were retrospectively analyzed using the framework developed in the theoretical part of the study to answer the following questions:

- What are game development processes like in terms of high-level process models?
- What are the needs of game development processes in terms of support areas?

To help the reader comprehend the context of this empirical part of the study an introduction to game development in general was provided.

As for the results of the case study, although in neither of the processes was an up-front decision made to use some specific high-level process model, both of them clearly had elements of specific high-level process models. As neither of the processes, however, represented a pure form of any of the existing high-level process models, it was concluded that approaches used in the examined processes represented some kind of adaptations of the existing models. It was also realized that although the resemblance was not due to up-front decision to use a specific high-level process model (existing or new), it was not accidental either. Instead, the resemblance was due to both intentional gradual adaptation to but also careful anticipation of the emerging needs.

The analysis of the role of the support areas, then, gave indications on the reasons as to why, in particular, did the examined game development processes resemble specific high-level process models. To sum up, it was found that operational software played an important role as a progress sign and in requirements elicitation, and that software modifiability and clarity were vital especially in customer-specific process due to intense requirements evolution. Further, it was discovered that comprehensive documentation was considered one of the process end results although continuous updating activities during the development were not seen feasible. It was also found out that risk management at the single-project level did not play much of a role in either of the processes.

All in all, the two game development processes examined were very different in nature. As the first game development process (HDML) was about new product development beginning from the concept development and ending with release of preliminary product version, the second one (SMS) was about tailoring an existing preliminary product version to meet operator-specific needs. Further, the HDML software process was purely internal, whereas operator-specific SMS process, by definition, applied to customer projects. This internal

versus external nature of the processes was reflected in their planning orientation versus experimentation orientation respectively: whereas in the internal software process it was possible to act based on thorough preplanning, the customer-specific process required iteration rounds based on customer experimentation. Thus, the two processes showed that straightforward and planning-oriented approach might not be feasible if requirements engineering essentially involves actors that are not familiar with the product concept. In these cases, experimentation and iteration are needed to elicit and meet the customer requirements. Further, in quick-paced projects the iteration has to be combined with an adequate level of control to ensure quick and easy iteration.

The results of the case study are consistent with the previous discussion on the high-level process models, that is, there is a homeground for both experimentation-oriented and orderly planning-oriented (or even linear) approaches (Boehm 2002). Experimentation-oriented approaches excel in environments where requirements are not pre-specifiable, orderly planning-oriented approaches, in turn, work best when the requirements and other important factors are relatively stable and can be determined in advance. Moreover, in situations where creativity is required (typical of game development) rigidity in software process definition should be avoided in order to make flexible tailoring of the software process to the unique needs of the project possible (Fayed 1997; Armour 2001; Cameron 2002).

As for the contributions of this study, one of these is the development and "piloting" of the framework that can be used by researchers as well as practitioners for software process analysis in terms of high-level process models and support areas. This framework can be used in game development as well as any other software development. By developing this framework based on a comprehensive review on the high-level process models, this study also aimed to consolidate the theoretical basis for the high-level process models, which until now has remained incoherent. The study also aimed to help establish basis for the further game development research, by bringing out the growing importance and complexity of the game development processes and by conducting a pilot study in this area.

The limitations of this study are as follows. The case study was restricted to one company. Further, two mobile game development processes were explored based on interviewing one person representing the company. Consequently, the findings of the case study cannot be reliably generalized to other environments. However, it should be noticed that the express

goal of this study was to give an overview picture of game development processes in one mobile game development company, and thereby help establish a basis for further game development research. Limiting the number of the cases made it possible to form a more thorough picture of the research object. This was taken into consideration in the interviewee selection in which the possibility of composing an adequate overview of the software processes was emphasized.

Some limitations to the case study are also due to using only one method for data collection and thus the lack of triangulation. However, the data collection was conducted with in-depth interviews and the interviewee was later consulted to assure the accuracy of the interpretation. Further, combining the interviews with other data sources like, for example, direct observation of the examined game development processes in their entirety would probably have provided the researchers with more detailed picture of the research object but was not feasible due to access limitations. Finally, as for the framework developed in the theoretical part of the study (and used in the empirical part), this is only at its preliminary stage and should be treated accordingly. Especially the part of the framework concerning the support areas should be further refined and augmented in future research. Further, when examining the suitability of this framework for high-level process model selection and creation, the possibilities of combining the framework with other high-level process model selection tools like the selection criteria proposed by (Alexander and Davis 1991) or spiral model proposed by (Boehm 1989) deserve to be examined.

Other issues not within the scope of this study but deserving to be recognized in the future research are:

- How was the project success assessed in the game development processes examined in this case study? To further deepen the view now given about the two development processes it would be useful to know how the successfulness of the described processes was assessed and further, how the processes succeeded. In other words, the processes used specific approaches but with what kind of results?
- How do the game development processes related to massive and complex pc/video games (or more sophisticated mobile games) and relatively small and simple mobile games described in this study differ from each other? A comprehensive study should be conducted to examine the development processes related to both mobile and other

games to both further refine the framework developed in this study and to get a more comprehensive overview of the game development processes.

- How much are software process definition and modeling used in game development companies? What are the experiences of using software process definition and modeling in games industry? Are there differences in the use process definition and modeling between the game development companies that are of different size or age? How does game development project size (measured in time, scope, persons, et cetera) relate to the use of software process definition and modeling? Addressing these questions would help get an overview picture of the systematicness of the game development processes and would also give indications of reasons affecting it.
- Finally, it would be useful that future research would focus on further solidifying the theory base for existing high-level process models. Especially, it should be examined if (and how) these models are really used in practice and with what results. The bottom line is that before inventing new models, we should help practitioners use the existing ones (Wiegers 1998). Process models alone are not enough, but people are needed to apply them, at the right time in right proportions (Bach 1995).

There is, obviously, a lot of research to do if an understanding of the ever-growing game development industry from the software process point of view is striven for. An increased understanding of the game development processes as software processes may be needed to help game development companies succeed in their ever more complex undertakings. In addition to helping the game development companies to manage their work better the increased understanding of the game development processes would probably be of use for financiers and investors interested in this industry.

APPENDICES

Appendix 1: The general topics provided to the interviewee in advance 21.1.2003.

Interviewee Antti Laaksonen, software engineer, Codetoys, Inc.

- Pelikehitysprosessin päävaiheet, niiden sisältö ja tuotos
- Päävaiheiden ajallinen järjestys, perättäisyys ja päällekkäisyys
- Päävaiheiden iteroiminen
- Kokeilun avulla oppiminen versus kattava etukäteissuunnittelu
- Tuotteen kehittäminen yhtenä kokonaisuutena ilman väliversioita versus versioittain toiminnallisuutta asteittain lisäten
- Prototyyppien käyttö
- Tarpeiden määrittelyn ja dokumentoinnin toteutus
- Ohjelmiston rakenteen suunnittelun ja dokumentoinnin toteutus
- Johdon käyttämät keinot pelituotantoprojektin edistymisstatuksen arvioimiseen
- Asiakkaan, julkaisijan, rahoittajan tms. käyttämät keinot pelituotantoprojektin edistymisstatuksen arvioimiseen
- Riskienhallinta
- Lopputuotteen ylläpitäminen julkaisemisen jälkeen
- Komponenttien uudelleenkäyttö
- Vaihejakomallien (elinkaarimallit, life cycle mallit) käyttö pelituotannossa

Appendix 2: The interview guide used in the interview 30.1.2003.

Interviewee Antti Laaksonen, software engineer, Codetoys, Inc.

Haastattelun tarkoitus:
Ajatus taustalla on seuraava: eri tuotteet vaativat erilaisia kehitysprosesseja, erilaiset kehitysprosessit edustavat erilaisia vaihejakomalleja. Haastattelulla halutaan selvittää millainen on pelikehitysprosessin luonne (millaista vaihejakomallia edustaa), ja millaista tukea tämänluonteinen prosessi tarvitsee (millaisella vaihejakomallilla pelikehitysprosessia voi tukea). Haastattelu on rakenteeltaan varsin avoin ja pyrkii vastaamaan tutkimuskysymyksiin pyytämällä haastateltavaa kertomaan seuraavista asioista:

Taustatiedot:

Yritys	Nimi	
	Ikä	
Projekti	Nimi	
	Ajankohta	
	Kesto	
	Muuta	
Projektin tuote	Nimi	
	Genre	
	Lyhyt kuvaus	
	Muuta	
Haastateltava	Nimi	
	Asema organisaatiossa	
	Asema projektissa	
	Haastattelupäivämäärä	

I Yleiskuva:

Kohdan I kysymyksillä pyritään muodostamaan kuvaa pelikehitysprosessin rakenteesta ja luonteesta korkealla tasolla.

Minkälaisia päävaiheita prosessissa oli:

- Mitkä olivat mielestäsi päävaiheet?
- Mikä oli kunkin vaiheen sisältö lyhyesti?
- Päävaiheiden ajallinen järjestys mahdolliset ajalliset päällekkäisyydet ja iteraatio?
- Miksi iteroitiin (vanhojen korjailu, uusien elementtien tekeminen)
- Miten päätettiin että tuote on valmis?

Tuotteen rakentamisen vaiheistus:

- Kehitettiinkö tuote lineaarisesti yhtenä kokonaisuutena ilman väliversioita määrittelystä toteutukseen vaiko pikemminkin toisiinsa integroitavina paloina tai ydinrungon päälle toiminnallisuutta kierroksittain lisäten?
  - Minkälaisiin eri väliversioihin tai paloihin kehitys jakaantui?

- Päätettiinkö jo projektin alussa millaista toiminnallisuutta jen. Kuhunkin väliversioon/palaan tulee vai tarkentuiko tämä toteutuksen aikana
- Kehitettiinkö eri väliversioita/paloja ajallisesti rinnakkain vai peräkkäin?
- Olivatko väliversiot laatu-ja suoritustasovaatimuksiltaan sellaisia että tuote olisi voitu tarpeen tullen toimittaa asiakkaalle riisuttuna versiona?

- Pyrittiinkö tuotteesta saamaan aikaiseksi toiminnallinen esi/ydinversio jo aikasiessa vaiheessa? Miksi?

- Aloitetttiinko tuotteen kehittäminen nollasta vai liittyikö prosesssiin uudelleenkäyttöä? Käytettiinkö jotain, jota voidaan käyttää uudelleen tulevaisuudessa/on käytetty aiemminkin jossain peliprosesseissa?

II Projektin sisältöä tarkentavat kysymykset:

Kohdan II kysymyksillä pyritään tarkentamaan kuvaa pelikehitysprosessin rakenteesta ja luonteesta.

Idean määrittely ja dokumentointi (concept development, esitutkimus):
- Mistä pelin idea/konsepti syntyi
- Millaista dokumentaatiota konseptiin liittyi
- Mihin konseptin dokumentoimista tarvittiin

Vaatimusmäärittely (functional specification):
- Päätettiinkö ennen toteutusta selkeästi mitä asioita peliin tulee
- Muuttuivatko tai tarkentuivatko vaatimukset asteittain tuotantoprojektin myötä vai pysyttiinkö alussa päätetyissä?
- Millaista dokumentaatiota vaatimuksiin liittyi?
  - Missä vaiheessa vaatimusdokumentaatio tehtiin?
  - Mikä oli vaatimusdokumentaation rooli pelikehitysprosessissa?

Tekninen suunnittelu ja dokumentointi (technical specification):
- Miten rakenne ja tekninen toteutus suunniteltiin ennen varsinaista toteutusta vai lähdettiinkö koodaamaan ilman kattavampaa suunnittelua?
- Millaista teknistä dokumentaatiota oli:
  - Missä vaiheessa tekninen dokumentaatio tehtiin?
  - Mikä oli teknisen dokumentaation rooli pelikehitysprosessissa?

- Muuttuiko rakenne suunnitellusta prosessin aikana?
  - Mitkä asiat tähän vaikuttivat?
  - Miten tällaisia rakenteeseen liittyiviä muutoksia pyrittiin hallitsemaan?

- Pyrittiinkö rakenteen suhteen muokattavuuteen esim. modulaarisuuden ja lähdekoodin siisteyden kautta?

Testaus:
- Mitä vaiheita testauksessa oli?
- Missä vaiheessa ajettiin peli organisaation ulkopuolelle beta testaukseen? Mitä tässä ilmeni?

Käytettiinkö prototyyppejä? Missä vaiheessa ja miksi? Varsinaisesta lopputuotteesta erillisiä tai lopputuotteeseen sisällytettäviä?

Täydennettiinkö tuotetta ensimmäisen julkaisemisen jälkeen jotenkin?

Projektin johtaminen:

- Miten oma johto arvioi pelituotantoprojektin edistymistä/tilaa?
- Miten asiakas, julkaisija, rahoittaja tms. ulkopuolinen taho arvioi pelituotantoprojektin edistymistä/tilaa?
- Miten projektin riskit kartoitettiin? Kuinka ne vaikuttivat projektin ohjaukseen?
- Onko yrityksessä pyrkimystä monistaa jotain tiettyä toimintatapaa eri projektien välillä? Onko tätä varten käytössä projektikäsikirja tms. projekti/prosessiaineistoa?

III Vaihejakomallin käyttö:

Kohdan III kysymys kartoittaa vaihejakomallien käyttöä pelituotantoprosessissa. Kartoitetaan miten organisaatio itse näki tilanteen: käytettiinkö jotain mallia, mitä. Verrataan vastausta kohdan i ja ii vastauksiin ja katsotaan vastasiko käsitys kuvattua.

- Oliko määritelty käytettäväksi joku tietty korkean tason vaihejakomalli?

ZOBMONDO!!

the outrageous game of bizarre choices™

for 3 or more adult players or teams

Note: Although some ZOBMONDO!!™ questions may paint a risqué, nauseating, or downright disturbing picture, they are NOT to be taken literally (don't try any of them!), nor are they meant to offend. Instead, they are designed to make you think and engage in lively discussion. Feel free to skip questions that might be too vivid for your particular group, or go ahead and destroy a card if you find it too offensive. Then place yourself in the proper fun, social state of mind and have a great time playing!

Contents

- 300 cards
- 5 colored gems
- 1 drawstring pouch
- 1 ballot pad
- 1 score pad

Setup

- The score pad will serve as the gameboard. Choose a scorekeeper for the game and give the score pad to that player.
- Put the colored gems into the drawstring pouch.
- Place the ballot pad within easy reach of all players.

OBJECT

Predict how other players will answer questions and be the first player to have your initials written in all the squares on the score pad.

GAMEPLAY

The oldest player goes first. Play passes to the left.

On Your Turn:

- Draw a card from the box. You are now known as the "Zobber."
- Draw a gem from the pouch and match its color to the corresponding question on the card (see chart). Silently read the question and try to predict which answer the other players will choose. Write the answer on a ballot and place it facedown in front of you.
- Read the question out loud to the other players.
- Players discuss which of the question's two options they prefer. The goal is **consensus**—see the definition above as it applies to gameplay.

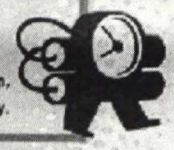
Note: No matter how disgusting, obnoxious, heinous, or hilarious the choices are, saying "I don't like either" is not allowed in ZOBMONDO!! Players must choose one of the options.

- When everyone has agreed, a member of the group announces which answer has been chosen.
- Turn the ballot faceup.
- If you incorrectly predicted the answer, you must wait until your next turn to try again.

The gems and cards are color coded. Match the gem to the corresponding category on the card:
Yellow ... Pain/Fear/Discomfort
Red ... Appearance/Embarrassment
Purple ... Food/Ingestion
Blue ... Ethics/Intellect
Green ... Random

CONSENSUS: A group decision. In ZOBMONDO!!, consensus is reached after a lively, uninhibited discussion about which of the two bad options for a question is better. Consensus can be reached by voting, discussing or debating. Use whatever process you choose, but don't get hung up on it. Try to take no more than 3 minutes to come to an agreement.

Players who purposefully prevent the group from reaching consensus should be labeled "difficult" and banished from gameplay. The goal of ZOBMONDO!! is entertaining social interaction, not competitive game strategy.

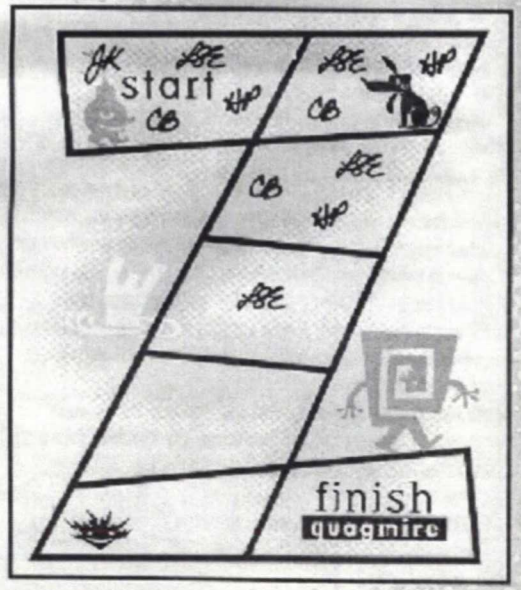


- If you correctly predicted the answer, the scorekeeper writes your initials on the first square of the score pad, and your turn is over.

Note: Your initials in a square act as your "mover" as you race towards the Finish. You cannot leave a square and move along the score pad until, on your turn as Zobber, you earn your initials in that square by correctly guessing your opponents' answer.

- Play passes to the left.

The Score Pad



The Quagmire™

The QUAGMIRE appears at the end of the score pad. Successfully complete your turn in this space, and you win the game! When you reach the QUAGMIRE:

- Invent your own question. Your goal is to ask a question that makes it difficult for other players to easily choose one option. In the QUAGMIRE, you want to avoid consensus!
- Read your invented question to the group.
- Players vote separately and write their individual choices on ballots. No discussion is permitted.
- Collect ballots and read the individual choices out loud.
- If all players have chosen the same answer, you are stuck in the QUAGMIRE and must wait until your next turn to invent a new question and try again.
- If players have chosen different answers, the scorekeeper writes your initials in the QUAGMIRE square and you win.

SPECIAL NOTES ON Quagmire GAMEPLAY:

- Do not use questions you've already heard or read. This is a major violation and is considered cheating (unless you choose ahead of time to ignore the QUAGMIRE — see "A Variation on QUAGMIRE Gameplay"). Be original and have fun making up your own questions.
- Stay true to the spirit of ZOBMONDO!! Asking others a question like, "Would you rather drive a white car -OR- a red car?" in order to get a divided response is also a major violation.
- Players may challenge a Zobber if either violation occurs. If the group agrees on the violation, the Zobber must come up with another original question or lose the turn.

A VARIATION ON Quagmire GAMEPLAY

If you think making up an original question will be too difficult, all players can agree at the beginning of the game to ignore the QUAGMIRE rules and continue regular ZOBMONDO!! gameplay in the QUAGMIRE square instead.

Team Play

It's more efficient to play with teams if you have more than 8 players. One team "Zobs" by quickly and silently agreeing on which answer the other teams will choose. In the QUAGMIRE, the Zobbing team makes up a question, and the other teams agree silently on their answer before writing it on a ballot. Consensus becomes even more important with team play.

WINNING

The first player to reach the final QUAGMIRE space on the score pad and have his or her initials written there wins.

To order additional ballot and score pads, send a check made out to HPD for $3.50 (includes postage, handling and 2 sets of pads). Mail to HPD, P.O. Box 693, Pawtucket, RI 02862, or telephone 888-836-7025 (toll-free) to place a Visa or MasterCard order. California and Rhode Island residents, add sales tax. Please allow 4–6 weeks. Offer valid in U.S. while supplies last, and price is subject to change without notice. We reserve the right to limit quantities.

Proof of Purchase

ZOBMONDO!!

41406-1 022900

INDEX OF FIGURES AND TABLES

REFERENCES

(2001). "The UK Games Industry and Higher Education". Human Capital - Media Strategy and Research, Final Report.

(2001a). "Economic Impacts of the Demand for Playing Interactive Entertainment Software". Interactive Digital Software Association (IDSA), Report.

(2001b). "State of the Industry Report 2000-2001". Interactive Digital Software Association (IDSA), Report.

(2002). "2001 Game Industry Sales Data & Graphs." 19.3. 2002. http://www.idsa.com/pressroom.html.

(2002). "Games Investor." 12.3. 2002. http://www.gamesinvestor.co.uk.

(2002). "NPD Techworld Reports Improved Software Sales During First Half '02: New Operating Systems, Virus Protection and Games Software Drive Growth." 1.12. 2002. http://www.npd.com/.

Abrahamsson, P., O. Salo, et al. (2002). "Agile software development methods: Review and analysis". VTT, publications 478.

Alexander, L. and A. Davis (1991). Criteria for the Selection of a Software Process Model. 15th IEEE International Conference on Computer Software and Applications (COMPSAC), Tokyo, Japan.

Andriole, S. J. (1994). "Fast cheap requirements: prototype or else." IEEE Software(March): 85-87.

Anjard, R. P. (1995). "Management and planning tools." Training for Quality 3(2): 34-37.

Armour, P. G. (2001). "Matching Processes to Types of Teams." Communications of the ACM 44(7): 21-23.

Bach, J. (1995). "Enough about process: what we need are heroes." IEEE Software 12(2): 96-98.

Bandinelli, S., A. Fuggetta, et al. (1995). "Modelling and improving an industrial software process." IEEE Transactions on Software Engineering 21(5): 440-454.

Basili, V. and A. Turner (1975). "Iterative enhancement, a practical technique for software development." IEEE Transactions on Software Engineering se-1(4): 390-396.

Bates, B. (2001). Game Design: The Art and Business of Creating Games, Prima Tech.

Benbasat, I., D. K. Goldstein, et al. (1987). "The Case Research Strategy in Studies of Information Systems." MIS Quarterly 11(3): 368-386.

Benington, H. (1983). "Production of Large Computer Programs." Annals of the History of Computing 5(4): 299-310.

Bersoff, E. H. (1991). "Impacts of Life Cycle Models on Software Configuration Management." Communications of the ACM 34(8): 104-118.

Blackburn, J., G. Hoedemaker, et al. (1996). "Concurrent software engineering- prospects and pitfalls." IEEE Transactions on Engineering Management 43(2 May): 179-188.

Boehm, B. (1989). What we really need are process model generators. Proceedings of the 11th International Conference on Software Engineering, Pittsburg (PA), USA.

Boehm, B. (2000). "Spiral Development: Experience, Principles, and Refinements". Software Engineering Institute, Spiral Development Workshop 2000, Special report, CMU/SEI-2000-SR-008.

Boehm, B. and F. Belz (1990). Experiences with the spiral model as a process model generator. Proceedings of the 5th International Software Process Workshop on Experience with Software Process Models, Kennebunkport, Maine, USA.

Boehm, B., A. Egyed, et al. (1998). "Using the winwin spiral model, a case study." IEEE Computer 31(7): 33-44.

Boehm, B., T. Gray, et al. (1984). Prototyping vs. specifying. Proceedings, 7th International Conference on Software Engineering, Orlando, Florida, USA.

Boehm, B. W. (1976). "Software engineering." IEEE Transactions on Computers c-25(12 December): 1226-1241.

Boehm, B. W. (1979). Software engineering - as it is. Proceedings of the 4th International Conference on Software Engineering, Munich, Germany.

Boehm, B. W. (1987). Software process management: lessons learned from history. Proceedings of the 9th International Conference on Software Engineering, Monterey (CA), USA.

Boehm, B. W. (1988). "A spiral model of software development and enhancement." IEEE Computer 21(5): 61-72.

Boehm, B. W. (2002). "Get Ready for Agile Methods, with Care." IEEE Computer 35(1): 64-69.

Brooks, F. (1987). "No silver bullet: essence and accidents of software engineering." IEEE Computer 20(4): 10-19.

Brooks, F. P. (1982). The Mythical Man-month: Essays on Software Engineering, Addison-Wesley.

Cameron, J. (2002). "Configurable Development Processes- Keeping the Focus on what is being produced." Communications of the ACM 45(3): 72-77.

Chroust, G. (1996). "What is a software process." Journal of systems architecture 42(8): 591-600.

Clements, P. C. (1995). "From subroutines to subsystems, component based software development." American programmer 8(11): 1-8.

Crabtree, S. (2000). "Killing Feature Creep Without Ever Saying No." 28.10. 2002. http://www.gamasutra.com/features/20001020/crabtree_01.htm.

Crosby, O. (2000). Working so others can play: Jobs in video game development. Occupational Outlook Quarterly. 44: 2-13.

Curtis, B., M. I. Kellner, et al. (1992). "Process modeling." Communications of the ACM 35(9 September, Special issue on analysis and modeling in software development): 75-90.

Davis, A. M. and P. Sitaram (1994). "A Concurrent Process Model of Software Development." ACM Software Engineering Notes 19(2): 38 - 51.

Davis, G. (2000). Write is Might. Game Developers Conference (GDC), San Jose, CA, US.

Derniame, J.-C., A. K. Badara, et al., Eds. (1999). Software Process: Principles, Methodology, Technology. Lecture Notes in Computer Science, Springer.

Dyer, M. (1980). "The management of software engineering part 4: Software development practices." IBM Systems Journal 19(4): 451-465.

Fairley, R. (1985). Software engineering concepts. New York (NY), McGraw-Hill.

Fayed, M. E. (1997). "Software Development Processes: A Necessary Evil." Communications of the ACM 40(9): 101-103.

Fowler, M. (2001). Agile Manifesto. Software Development.

Fuggetta, A. (2000). Software Process: A Roadmap. ICSE 2000, 22nd International Conference on Software Engineering, Future of Software Engineering Track, Limerick, Ireland.

Gibbs, W. W. (1994). "Software's Chronic Crisis." Scientific American(September).

Gilb, T. (1985). "Evolutionary delivery versus the waterfall model." ACM Software Engineering Notes 10(3): 49-61.

Gilb, T. (1988). Principles of software engineering management, Addison-Wesley.

Gladden, G. R. (1982). "Stop the life cycle-I want to get off." ACM Software Engineering Notes 7(2): 35-39.

Gordon, V. and J. Bieman (1995). "Rapid prototyping, lessons learned." IEEE Software(Jan): 85-95,.

Haikala, I. and J. Märijärvi (2000). Ohjelmistotuotanto 7. painos. Helsinki, Satku-Kauppakaari Oyj.

Hannabuss, S. (1996). "Research interviews." New Library World 97(1129): 22-30.

He, Z., G. Staples, et al. (1996). Fourteen Japanese quality tools in software process improvement. The TQM Magazine. 8: 40-44.

Highsmith, J. and A. Cockburn (2001). "Agile Software Development: The Business of Innovation." IEEE Computer 34(9): 120-122.

Humphrey, W. (1998). "Three Dimensions of Process Improvement. Part I: Process Maturity." The Journal of Defense Software Engineering(February).

Humphrey, W. and M. Kellner (1989). Software Process Modelling: Principles of Entity Process Models. Proceedings of the 11th International Conference on Software Engineering, Pittsburg (PA), USA.

Humphrey, W. S. (1989). "CASE Planning and the Software Process". Software Engineering Institute, Technical Report, CMU/SEI-89-TR-26.

Humphrey, W. S. (1989). Managing the software process. Reading (MA), Addison-Wesley.

IEEE (1990). "IEEE Std 610.12-1990 Standard glossary of software engineering terminology".

Iivari, J. (1990). "Hierarchical spiral model for information system and software development. Part 1: theoretical background." Information and software technology 32(6): 386-399.

Iivari, J. (1991). "A paradigmatic analysis of contemporary schools of IS development." European Journal of Information Systems 1(4): 249-272.

Iivari, J. and E. Koskela (1987). "The PIOCO model for Information Systems design." MIS Quarterly 11(3): 401-419.

Isoda, S. (1991). An experience of software reuse activities. Fifteenth IEEE International Conference on Computer Software and Applications (COMPSAC), Tokyo, Japan.

Kellner, M. I. (1988). Representation formalisms for software process modelling. Proceedings of the 4th International Software Process Workshop on Representing and Enacting the Software Process, Devon, UK.

Lehman, M. (1998). "Software's Future: Managing Evolution." IEEE Software 15(1): 40-44.

Lehman, M. M. (1987). Process models, process programs, programming support. Proceedings of the 9th International Conference on Software Engineering, Monterey (CA), USA.

Lehman, M. M. (1997). Process Modelling - Where Next. Proceedings of the 19th International Conference on Software Engineering, Boston, Massachusetts, USA.

Linger, R. C. (1980). "The management of software engineering part 3: Software design practices." IBM Systems Journal 19(4): 432-450.

Lonchamp, J. (1993). A structured conceptual and technological framework for software process engineering. Proceedings of the 2nd International Conference on Software Process (ICSP); Continuous Software Process Improvement, Berlin, Germany.

McConnell, S. (1996). Rapid Development-Taming wild software schedules. Redmond (WA), Microsoft Press.

McCormick, A. (2001). "Programming Extremism." Communications of the ACM 44(6): 109-111.

McCracken, D. D. and M. A. Jackson (1982). "Life cycle concept considered harmful." ACM Software Engineering Notes 7(2): 29-32.

Meyer, M. and R. Seliger (1998). "Product platforms in software development." Sloan Management Review 40(1): 61-74.

Mills, H. D. (1976). "Software development." IEEE Transactions on Software Engineering se-2(4 December): 265-273.

Mills, H. D. (1999). "The management of software engineering-principles of software engineering." IBM Systems Journal -Turning Points in Computing: 1962-1999 38(2 & 3): 289-295.

Muzyka, R. (2000). The Making of a Monster: Creating Baldur´s Gate. Game Developers Conference (GDC), San Jose, CA, US.

Neumann, P. G. (1993). "System Development Woes." Communications of the ACM 36(10): 146.

Nierstratz, O., S. Gibbs, et al. (1992). "Component oriented software development." Communications of the ACM 35(9): 160-165.

Ohiwa, H., N. Takeda, et al. (1997). "KJ editor: a card-handling tool for creative work support." Knowledge-Based Systems 10(1): 43-50.

O'Neill, D. (1980). "The management of software engineering part 2: Software engineering program." IBM Systems Journal 19(4): 421-431.

Oster, T. (2000). Project prototyping,. Game Developers Conference (GDC), San Jose, CA, US.

Osterweil, L. (1997). Software Processes are Software Too, revisited: An invited talk on the most influential paper of ICSE 9. Proceedings of the 19th International Conference on Software Engineering, Boston, Massachusetts, USA.

Ovum (2001). "3G Survival Strategies: Build, Buy or Share." 4.3. 2002. http://www.ovum.com/research/.

Parnas, D. L. and P. C. Clements (1986). "A rational design process - How and why to fake it." IEEE Transactions on Software Engineering 12(2): 251-256.

Pressman, R. (2000). Software engineering -a practitioners approach, McGraw-Hill.

Quinnan, R. E. (1980). "The management of software engineering part 5: Software engineering management practices." IBM Systems Journal 19(4): 466-477.

Redwine, S. and W. E. Riddle (1988). Software reuse processes. Proceedings of the 4th International Software Process workshop on Representing and Enacting the Software Process, Devon, UK.

Royce, W. W. (1987). Managing the development of large software systems: concepts and techniques (republished). Proceedings of the 9th International Conference on Software Engineering, Monterey (CA), USA.

Ryan, T. (1999a). "The Anatomy of a Design Document." 28.10. 2002. http://www.gamasutra.com/features/19991019/ryan_01.htm.

Ryan, T. (1999b). "Controlling Chaos in the Development Process." 28.10. 2002. http://www.gamasutra.com/features/19990806/controlling_chaos_01.htm.

Schach, S. R. (2002). Object-Oriented and Classical Software Engineering, McGraw-Hill.

Schmidt, R., K. Lyytinen, et al. (2001). "Identifying Software Project Risks: An International Delphi Study." Journal of Management Information Systems 17(4): 5-36.

Sommerville, I. (2001). Software Engineering 6th edition, Addison-Wesley.

Thayer, R. H., Ed. (1997). Software engineering project management, 2nd edition. Los Alamitos (CA), IEEE Computer society.

Thayer, R. H. (1997b). "Software Engineering Project Management". Software engineering project management, 2nd edition. R. H. Thayer. Los Alamitos (CA), IEEE Computer Society: 72-104.

Walton, G. (1998). "Bringing Engineering Discipline to Game Development (Originally Published in Game Developer Magazine, December, 1998)." 28.10. 2002. http://www.gamasutra.com/features/production/19981218/eng_disc_01.htm.

Ward, R. P., M. E. Fayed, et al. (2001). "Software Process Improvement in the Small." Communications of the ACM 44(4): 105-107.

Weihrich, H. (1997). "Management: Science, Theory, and Practice". Software engineering project management, 2nd edition. R. H. Thayer. Los Alamitos (CA), IEEE Computer Society: 4-13.

Whitten, N. (1995). Managing software development projects, 2nd edition, John Wiley & Sons, Inc.

Wiegers, K. E. (1998). "Read My Lips: No New Models!" IEEE Software 15(5): 10-13.

Wirth, N. (1971). "Program development by stepwise refinement." Communications of the ACM 14(4 April): 221-227.

vom Scheidt, G. (2000). Data Management in Game Development Projects. Game Developers Conference (GDC), San Jose, CA, US.

Yin, R. K. (1990). Case Study Research: Design and Methods, Sage Publications.

Yu, E. S. K. and J. Mylopoulos (1994). Understanding "why" in software process modelling, analysis, and design. Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy.